National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

A Protocol for Distributed UNIX Shells

BY

Timothy W. Breitkreutz

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-88356-1

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Timothy W. Breitkreutz

TITLE OF THESIS: A Protocol for Distributed UNIX Shells

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1993

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) . . . . . .

Timothy W. Breitkreutz
11119 - 41 Avenue
Edmonton, Alberta
Canada T6J 0T2

Date: 7 OCT 1993 . . . . . .

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **A Protocol for Distributed UNIX Shells** submitted by Timothy W. Breitkreutz in partial fulfillment of the requirements for the degree of Masters of Science.

. . . . . .

Dr. T.A. Marsland (Supervisor)

. . . . . .

Dr. John Samson (External)

. . . . . .

Dr. Pawel Gburzyński (Examiner)

Dr. W.W. Armstrong (Chair)

Date: 7 OCT 1993

To the Liberation of Tibet

Abstract

A protocol for shell programmes to harness distributed UNIX workstation computing resources on a network is proposed and evaluated. The protocol allows a shell, graphical desktop, or scripting language to execute its subcommands on other computers by contacting them as servers. In a decentralised fashion, servers only respond to requests that they are capable of fulfilling, preventing the need for a centralised database, load-balancing system, or capability service. Faster and more available servers inherently respond more quickly to service requests, improving the performance of the distributed system. Furthermore, a distributed shell allows users to ignore which system can run different commands, since only servers capable of executing a command by a given name will respond. The proposed protocol fills a gap between existing remote execution protocols, which are limited in their functionality, and fully distributed programming paradigms. The original UNIX philosophy of using a shell programme to piece together small programmes, thereby gaining flexibility and performance, is applied to the distributed environment.

Acknowledgements

I would like to express my gratitude to Dr. Tony Marsland, for being a great teacher, supervisor, and employer over the years. Thanks also to the other examiners, Dr. Gburzyński and Dr. Samson, for their helpful suggestions. Richard McFarlane, Helen Moritz, and Christian Walters were of invaluable help with proofreading.

For encouragement and support, thanks to my mother, father, sisters and brothers.

For many hours of talking, providing excellent distraction, and helping me work on my 1993 New Year's resolution I am endebted to the members of the O.S.C.F.S. and all other friends. In particular, Lori Kiel and Mark Hughes went far beyond the call of duty and their generosity will be remembered.

Thanks also to the Alberta Department of Environmental Protection for generously allowing me to use their equipment and to Ron Orr and Curt Romaniuk for providing leave (and encouragement) for this research work.

UNIX and UNIX System V are trademarks of Unix Systems Laboratories, Inc. SunOS is a registered trademarks of Sun Microsystems, Inc. SPARC and SPARCstation are registered trademarks of SPARC International, Inc. IBM, RISC System/6000, and AIX are registered trademarks of International Business Machines Corporation. DEC, AXP, and Alpha AXP are registered trademarks of Digital Equipment Corporation. CLIX and Intergraph are registered trademarks of Intergraph Corporation. Open Software Foundation, OSF and OSF/1 are trademarks of Open Software Foundation, Inc. X Window System is a trademark of Massachusetts Institute of Technology. AT&T is a registered trademark of American Telephone and Telegraph Company.

Contents

1 Introduction **1**

1.1 Terminology, Notation, and Network Sources 3

2 A Review of Shells and Distributed Computing **5**

2.1 Existing Shells and Desktops 5

2.2 High-Level Generic Distributed Computing 8

2.2.1 Socket Programming 10

2.2.2 Message-Passing Systems 11

2.2.3 Remote Procedure Calling 14

2.2.4 *Ad Hoc* Distributed Computing 17

2.2.5 Load-Sharing Systems 18

2.2.6 Determination of Idle Workstations 19

2.2.7 Graphical Application Builders 21

2.2.8 Process Migration and Checkpointing 22

2.2.9 Fully Distributed Operating Systems 23

2.2.10 Security 25

2.3 Distributing the Shell 27

3 Design of the Distributed Shell Protocol **28**

3.1 Design Considerations and Constraints 28

3.2 Interaction of Client and Servers 31

3.2.1 Advertisements 34

3.2.2 Job Application 38
3.2.3 Selection and Job Offer 39
3.2.4 Job Confirmation 41
3.2.5 Job Completion 41
3.3 Pipeline Setup 42
3.4 Signals and Job Control 45
3.4.1 Signal Messages 46
3.5 Security 47
3.6 Control and Error Recovery 48
3.6.1 The Quit Message 49
3.7 Checkpointing Service 49
3.7.1 Implementation 50
3.7.2 Checkpoint Messages 50
3.7.3 Checkpoint Acknowledgement Messages 51
3.8 Processor and Resource Classes 52
3.9 Interactive Commands 53
3.10 Files 53
3.11 Known Limitations 54

4 Implementation and Results **56**

4.1 Extension to Shell Function 56
4.2 Implementation Details 58
4.2.1 Built-in Commands 59
4.2.2 Input Syntax 60
4.2.3 Command Database 61
4.2.4 Command Pipelines 62
4.2.5 Server Operation 63
4.2.6 Execution of Subtasks 64
4.2.7 Input/Output Subtask 65
4.3 Performance of Distributed Shell 66

5 Conclusion 70

Bibliography 72

A Dish Message Format 76

B Example Startup Messages 79

List of Tables

3.1 Dish Control Message Types 33
3.2 Advertisement Message Fields 35
3.3 Application Message Fields 38
3.4 Offer Message Fields 40
3.5 Confirmation Message Fields 41
3.6 Completion Message Fields 42
3.7 Signal and Acknowledgement Message Fields 46
3.8 Quit Message Fields 49
3.9 Checkpoint Message Fields 51
3.10 Checkpoint Acknowledgement Message Fields 52

List of Figures

1 Client and Server Relationship 29

2 Interprocess Communication Channels 32

3 Pipeline Setup 43

4 Network 1 66

5 Network 2 67

Chapter 1

Introduction

Computing resources are becoming inexpensive, causing substantial change in how they are used to solve problems. The most rapid change occurring is the growth of processing and network resources. Networks of hundreds to thousands of workstation computers running a UNIX operating system are common, but unused network bandwidth is available and processors sit idle much of the time. There are many ways to approach the problem of making these resources available, and eventually, programming will focus on the network instead of the processor as it does now.

The study of harnessing these resources can be described as *generic distributed computing*. Other fields of distributed computing involve specialised hardware and networks designed specifically for certain applications. *Parallel computing* normally refers to tightly connected banks or pools of processors located in supercomputers.

Most human-computer interaction is controlled by a high-level command interpreter (shell) or a high-level graphical desktop. Current desktops and shells are still designed without provision for using computing resources over the network.

It is possible to design a protocol for distributing tasks on a computer network that will at once provide easy access to network computing resources and improve general performance within the existing shell or desktop framework. In this thesis, such a protocol is designed for the UNIX workstation environment and a partial implementation is tested to show its feasibility.

The proposed distributed shell (*dish*) protocol fills a gap between remote execution protocols such as *rlogin*, more sophisticated programming tools such as Sun Microsystems' Remote Procedure Call (RPC) [1] and eXternal Data Representation (XDR), new programming paradigms like ISIS [2] and PVM [3], and graphical paradigms such as *HIGHLAND* [4] and *Enterprise* [5, 6]. It takes advantage of the inherent concurrency in the UNIX shell design to provide distributed computing at the shell level. Currently, there are no protocols officially documented in the Internet Protocol Request for Comments (RFCs) for this kind of service [7].

Dish extends the functionality of the existing *rlogin* and *rexec* protocols by providing a mechanism for dynamically assigning tasks to machines, by making the distribution of work transparent to the user, and by providing error recovery procedures. It provides a network capability for UNIX shells similar to the windowing and graphic display service provided by the X window system [8]. Both the User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) layers of the Internet Protocol (IP) [9] are utilised to distribute the shell's subtasks.

For many applications, it is simpler to use a distributed shell with the existing standard UNIX input/output (I/O) library, or even lower-level system calls such as *read* and *write*, than to go to the expensive overhead of setting up a full RPC application. Furthermore, *dish* takes advantage of and encourages the traditional UNIX code re-use methodology of breaking up functionality into small programmes (*filters*) to increase their general use and versatility.

Finally, a distributed shell allows more efficient use of networks by reducing the number of interactive remote login sessions, which are wasteful of bandwidth because each keystroke is transmitted over the network.

This protocol can also be used by any application with similar requirements to shell scripts, such as *Perl* [10] and other user applications. For a graphical environment, the distributed shell protocol can be used to extend the functionality of desktops and window manager programmes.

The *dish* protocol uses a client/server approach and will be described through a

loose anthropomorphic analogy. *Dish* clients interact directly with users or their shell scripts. Each command pipeline given to the client is broken up into *jobs*, which are *advertised* over the network. Servers on the network *apply* for these jobs, and are given *offers* if they are considered the most appropriate for the task.

A distributed shell was implemented and tested on a heterogeneous network of workstations, and the performance shows that the idea of distributed subtasks in a shell environment may prove to be worthwhile as processors become faster and network bandwidth cheaper.

1.1 Terminology, Notation, and Network Sources

The protocol described in this thesis is called the *distributed shell protocol* or the *dish protocol*. The implementation described is also referred to as the *distributed shell* or simply as *dish*. An important term used throughout is *pipeline*. In this discussion, a pipeline will refer to a set of *commands* given to a shell that are linked together in a singly connected graph, where output from one command is fed as the input to the next. Commands refer to executable programmes on a UNIX system. These may be executable binaries for some type of processor, or executable shell scripts.

All versions of UNIX come with their own edition of the UNIX Reference Manual [11]. It is also available on-line through the `man` command. Each programme, system call, library routine, file format, and protocol is given a *manual entry*, also known as a *manual page*, though they often fill many pages. For convenience of finding the manual entries, references to them are given with the section number in parentheses. For example, [*umask(2)*] refers to the system call `umask()`, found in section two of the manual.

A typewriter font (e.g., `/bin/cat`) is used to denote input to or output from a computer system, and is used for defining the protocol's message formatting. It is also used to describe file and directory names in the UNIX environment.

Some references are to documents that are pending publication and were obtained

using the File Transfer Protocol programme [*ftp(1)*] from *anonymous FTP sites* on the *Internet*, a global network of connected computers using the IP protocol. Many sites on this network allow anonymous file transfers to distribute research papers, software, and other information. Also obtained via *ftp* were the protocol definition standards for the Internet (*Request for Comments* or RFCs). They are officially available from the Internet site `nic.ddn.mil`.

Chapter 2

A Review of Shells and Distributed Computing

The distributed shell protocol brings together two features of computing: shells and shell programming, and distributed computing. Shells are the top level of the UNIX environment and are well known and understood, so only a short overview will be presented here. Distributed computing, however, is a new and developing field. The distributed shell protocol deals with the highest level of distributed computing, that is, an interpreted language that builds on other programmes written in lower-level languages.

2.1 Existing Shells and Desktops

The UNIX operating system was originally designed [12] with a guiding philosophy of *divide and conquer*. Functionality was isolated and put in small pieces, called *filters*, which were invoked in various combinations using a *shell* programme. Responsibility for providing standard input, output and error channels either to keyboards and monitors or files was given to the *shell* programme. Code was reused in this manner and many operations on a UNIX system are accomplished by writing *shell scripts*, programmes written in the shell interpreted language, to harness the power of all

the small filter programmes and coordinate them to do a given job. This approach worked well, although twenty years after the development of UNIX, shells are no longer as widely used as they once were, and the operating system has diverged far from its original philosophies and has become something of a hulking giant. Shells are being partially replaced by graphical working interfaces (desktops) and new, complex scripting languages, such as Perl [10] and Tcl [13]. These alternatives to the shell can also be distributed using the *dish* protocol, but since they are less concurrent in their nature, they would not be able to gain as much advantage.

UNIX shell programmes allow the user to connect processes in a *pipeline*, meaning that the output from one command is placed on the input channel of the next command. An example is the standard UNIX method of removing duplicate lines from a file. First the file is sorted with the `sort` command, then piped into another command called `uniq`, which eliminates any repeated lines. Finally, if the output is to be viewed on a terminal, it is often finally piped into a *pager* such as `less`, which is responsible for letting the user read a page of output before it moves off the screen. The command line looks like:

```
sort < inputfile | uniq | less
```

Common shells include *sh(1)*, *csh(1)*, *ksh(1)*, and *bash(1)*, and come with an extensive manual entry in the UNIX Reference Manual [11] explaining the usage and features of each.

A main function of the shell is to find specific commands, though their location is not specified. UNIX has a hierarchical file system, consisting of directories and subdirectories to an arbitrary level. The top of the filesystem is a directory called `/`. Subdirectories are shown by a following `/` character. Therefore an explicit command is given with a name starting at the root directory, for example "`/usr/bin/cat`."

Users have *home directories*, which is where the shell typically begins executing [*login(1)*], and they may also have their own collection of executable files. When a shell encounters a command without a directory, it looks through a list of directories

where it expects executable files to reside. On UNIX this is traditionally located in an *environment variable* called `PATH`. The path may include an entry consisting of a '.', indicating the current working directory, instructing the shell to look also locally for commands requested. (In the UNIX filesystem, all directories have at least two directories in them, '.', which represents the directory itself, and '..', which represents its parent directory.) The process environment is a set of data that is inherited by all processes from their parents.

The UNIX filesystem was originally designed to be simple and small; however, networking and shared file systems have complicated the task of shell command execution. A traditional shell will look through the `PATH` and attempt to execute the first file that matches the command name specified, whatever its execution permissions and content. Once the shell invokes the system call to execute its subtask, the system call first checks the permission bits of the file [*chmod(2)*]. If the file is not marked as executable in the permission bits, the subshell execution fails. If the file is marked as executable, the system must check if it is a permissible executable for the host architecture. This is done by checking if the file begins with a *magic number* [*file(1)*, *a.out(5)*], or if it is an executable for an interpreted language such as a shell. Shell scripts begin with the character sequence `#!` followed by the name of the programme to interpret the commands, usually a shell.

Normally, all files in directories included in the `PATH` should be executable. Sometimes, however, other kinds of files are inadvertently placed in such directories, and in a heterogeneous network, some programmes will only work on a subset of the processors available to the user. Binary executables, in particular, will only run on a processor of the architecture type for which they were compiled. Shell scripts will only execute on a given processor if the appropriate shell is available on that system. This forces users either to have different executable command names for the different processors they use, or to use a different `PATH` for each machine they use, isolating executables common to the different environments. Another difficulty in extending the old shell paradigm to heterogeneous networks is provision for the same set of

system executables for each type of machine.

2.2 High-Level Generic Distributed Computing

In the past decade, high performance computers have changed from isolated processor-bound mainframes to small, numerous, networked multiprocessor systems.

Loosely coupled distributed processing began with networks of processors connected via serial lines. With the arrival of more advanced network technology such as ethernet, distributed processing came within the reach of most workstation application developers. However, there is still no generally accepted paradigm for developing distributed applications for these now common networks. Instead, several different approaches are becoming commonplace depending on the application, the network, and the background of the software developer.

The protocol framework discussed in this thesis consist of the United States' Defence Advanced Research Project Agency (DARPA) internetworking project, commonly called *TCP/IP*. This set of protocols has become a widespread and non-proprietary standard for networking, and is the main network protocol employed by UNIX and other open computing systems [9].

Fully generalised distributed operating systems are still not in widespread use. Systems like *Amoeba* may eventually render upper level protocols such as Remote Procedure Call (RPC), Telnet, and *dish* obsolete. However, there is a large installed base of systems using TCP/IP. There are over one million nodes, mostly UNIX, on the Internet alone. UNIX is a well-known operating system and many distributed processing ideas are gradually being added to it without a complete redesign of the system. *Dish* represents a linking step in the evolution from processor-bound to communication-bound computing. It attempts to make processing resources appear to be distributed in the same way that a fully distributed operating system would.

The earliest approach to distributed systems is still with us in various message-passing systems, which are well-suited to applications with little data sharing and

easy synchronisation. A more recent and popular methodology is derived from Sun Microsystem's Remote Procedure Call (RPC) package, and is particularly suited to developing applications on multi-threaded operating systems and for more difficult synchronisation problems. The current industry standards such as OSF/DCE (the Open Software Foundation's Distributed Computing Environment) are based on the remote procedure call paradigm [14]. A third general method is to use *ad hoc* means for distribution of processing work, with commands and facilities already available for other network operations, such as network file systems, and remote shell commands. *Load-sharing systems* provide one or both of the above communication facilities as well as a method for allocating processing resources in such a way as to balance the processing load across the network. A central issue for load-balancing or distributed applications is the resource allocation policy. Since these systems are being developed in a time when *graphical user interfaces* (GUIs) are the norm, they are being used to allow developers to visualise and connect distributed application elements. Sophisticated distributed systems allow for process checkpointing or migration to provide fault-tolerance and dynamic reallocation of resources, for example, when a workstation user wants the processor again after a network application has been using all available processor cycles. All these methodologies for work distribution are striving for the ultimate goal of an operating system based on the network, such as Amoeba.

The distributed shell provides an alternative to message-passing or remote procedure calling by extending the functionality of the shell or graphical desktop, and making use of already concurrent characteristics of these environments, thus extending the functionality of *ad hoc* distributed computing methods. It provides a step toward an environment that will provide users with processor resources in a fully generalised fashion. *Dish* has a checkpointing facility that can be used with general purpose and non-complex applications. Although not considered here, the protocol could be extended, especially on a homogeneous network, to include process migration.

Security is an integral part of distributed computing systems. Authentication

and privacy must be provided by any network system. Only isolated and simplistic applications do not need these facilities.

2.2.1 Socket Programming

The first implementation of TCP/IP in a UNIX system was provided by the Berkeley System Distribution (BSD) Version 4 UNIX *socket* interface, which has become a standard for UNIX networking and low level distributed implementations. This interface gives the programmer direct access to TCP and UDP. For the design of client/server protocols and services using BSD sockets, Comer and Stevens [15] offer useful guidelines. Another network programming model for UNIX is the more general *STREAMS* package [16] designed for System V, the commercial alternative to BSD UNIX. However, this software is proprietary and not supported on as many UNIX systems as the BSD socket interface.

The Berkeley socket interface provides a data abstraction called a *socket*, which appears in most respects like a UNIX *file descriptor* [*open(2)*]. The socket, once opened, can be used as an end-point for task-to-task communication. Two types of sockets are widely used on TCP/IP systems. One type uses TCP, and is known as a *stream* socket, since it behaves like a stream of bytes. Any number of bytes can be sent through a TCP socket at once, and the other side can read them at once or in smaller or larger pieces. TCP is a reliable protocol—all bytes are guaranteed to be delivered and in order, except for a catastrophic network or processor failure. The other type of socket, a *datagram socket*, uses UDP, which is actually a user-level packaging for the lower-level *Internet Protocol* (IP), which behaves in a similar fashion. Datagram protocols deliver packets of variable but limited length, and can lose, duplicate, or deliver them in a different order than they were sent. When these protocols were developed, it was believed that moving reliability to a higher protocol level would improve performance, since many networking technologies (e.g., ethernet connected with gateways) are not reliable, and also because they were designed as *internetwork* protocols that should not make any assumptions about the reliability of the networks

they are connecting.

Sockets can be *connected* to form a fixed communication link, so destination addresses do not need to be specified for each send and receive call. Connected sockets can be used almost exactly like open files or devices, and therefore can connect programmes that are not concerned about where input is coming from or where output is going. Indeed, in the UNIX environment, filters should not be aware of whether their I/O streams are connected to files, devices, or sockets.

To connect sockets, one end must be made *passive*, and the process controlling it must be set up to wait for incoming connection requests [*accept(2)*]. When a request comes in, a new socket is created and connected to the incoming request, thus allowing the first socket to be used for more incoming connections. The request comes from an active socket connection system call [*connect(2)*] from another process.

The passive socket does not need to know in advance where the connection is coming from, but the active side must know the *IP address* and *IP port* number on which to connect. IP addresses currently consist of a thirty-two bit value—a computer's globally unique address. For readability, they are normally expressed in decimal notation as four bytes separated by periods (for example, `129.128.220.8`). Port numbers are sixteen bit integers identifying different services and connections on the same host. Comer [9] provides more information on host and port addressing in TCP/IP.

To implement a distributed application at this level requires address lookup and socket management code. This can be cumbersome for complex applications. However, using this level allows a software developer to achieve optimal performance by choosing the most appropriate protocol for the application and the network, thereby avoiding unnecessary protocol layers.

2.2.2 Message-Passing Systems

Distributed systems can be categorised as tightly coupled or loosely coupled. Tightly coupled systems typically have processors sharing memory. Loosely coupled systems

have processors with their own localised memory. The original approach for loosely coupled systems was to connect them together with serial communication lines, and then pass messages over the lines from one processor unit to the other. Hwang *et al.* describe one of the early such systems [17]. Few people use serial lines for interprocess communication anymore, but the same communication paths are often simulated over networks.

Message-passing systems such as PVM [3], NMP [18] and ISIS [2] provide similar communication channels between processes. This paradigm is appropriate and natural for some applications but in general requires great effort for design and implementation. The primary limitation of the message-passing approach is the radically different programming paradigm required. Programmers must learn a new methodology, debugging becomes much more difficult, and existing sequential programmes must be redesigned and reimplemented. Calls to library routines to set up and control the communication channels must be inserted directly into the source code in order to use these message-passing systems; thus they represent a lower-level distributed paradigm.

The new methodology consists of reformulating algorithms to include various autonomous agents exchanging portions of the data they are working on. The size of the data being shared is limited by the performance of the communication lines or network. Often this limitation on sharing of data can lead to inefficiencies in algorithms. More importantly, though, timing considerations become important and seriously complicate the algorithm design process. Complexity of algorithms grows enormously when they are distributed asynchronously; that is, when pieces of work that are in some way dependent on each other are executing simultaneously on different computers, instead of each system waiting for the others to finish work they are dependent on.

Algorithms that take full advantage of message-passing communication tend to be asynchronous and non-deterministic, thus hampering the debugging phase of programme development. Non-deterministic programmes are those that do not necessar-

ily execute in a single, repeatable sequence. If the network being used is shared with other applications, as is assumed in the framework of generic distributed computing, timing of messages is influenced unpredictably. Depending on the protocol and amount of routing required, messages can be lost or delivered in a different order than they were sent. Therefore, to debug such a distributed programme, all timing possibilities must be taken into consideration. This may or may not be feasible, especially without a dedicated network. In some cases the complexity makes an application impossible to debug with currently available tools.

Finally, message-passing systems require programmes to be re-written or extensively modified. Code written for a sequential computer cannot take advantage of a distributed system automatically with this paradigm.

The Network Multi-Processor (NMP) is an example of a typical message-passing system. NMP service is registered on each participating machine. The master process is invoked on one machine and it starts by initialising the NMP. This initialisation procedure reads a configuration file to determine the intended destination of cooperating processes and communication paths between them. The application attempts to contact all specified machines through their NMP service daemon, and the configuration information is passed through to the daemon on each machine, where each start the process indicated in the configuration file. Once the communication paths are set up between all processes, they all exit from their respective calls to the NMP initialisation routine. Applications then communicate through these channels until their tasks are done and they call the shutdown library function, which cleans up all the tasks and closes communication channels.

Channels are simple TCP streams, and any data conversion is left to the user. Other message-passing packages enforce message typing, allowing programmers to avoid converting data from one machine's internal representation to another.

NMP configuration files can specify any arbitrary set of connections between processes. They can be fully connected, to simulate a bus, or they can be arranged in a tree, star, ring, or other hybrid fashion. The system has been enhanced by a dynamic

configuration file builder to allow run-time determination of machines that are currently idle. It can also add new communication channels between existing processes or start up new remote processes dynamically.

ISIS is another message-passing system. It uses the notions of *process groups* and *virtual synchrony*. Process groups are sets of processes that join through a membership service. Messages are then passed between process groups in a manner that is usually asynchronous but can be brought into synchrony for critical sections of code. This allows the system to gain performance improvements from asynchronous operations but still provide a degree of robustness and deterministic behaviour when these are needed. Virtual synchrony means that during certain communication operations, determinism is guaranteed.

ISIS also provides a more general purpose paradigm for message-passing implementations, that of multicasts to groups. It is intended to give processes the roles of *subscribers* and *publishers*, where information is flowing in from and out to variable sets of other processes. This allows for greater fault tolerance and ease of implementing some types of systems, the example given by Birman [2] being that of a brokerage firm trading stocks on multiple stock market.

The Parallel Virtual Machine (PVM) system [3] has become a popular message-passing system. It is similar to NMP, however, it provides data conversion over the communication channels with the use of XDR [19]. PVM provides only dynamic process creation, which is implemented via the *rexec(3)* library routine. It also provides a barrier mechanism to create critical sections where processes can be synchronised.

2.2.3 Remote Procedure Calling

Many programmers writing applications are turning to *multi-threaded* operating systems and remote procedure calls in combination to provide distributed processing. Sun Microsystems was the first to introduce remote procedure calls to UNIX with their RPC/XDR product [1]. Currently, the OSF's Distributed Computing Environment [14] has adopted this same approach.

Remote procedure calling extends the meaning of a regular procedure call within a programme. Normally, procedures allow communication between different parts of a programme executing as a single process. A remote procedure call takes the parameter information and sends a message to another process, which may be on a different processor and in a different address space, to execute it there. The result of the computation is then returned as the value of the function call. The main difference from regular procedure calling, therefore, is the lack of access to global memory space from within the procedure.

Remote procedure calls alone do not allow for distributed processing. The calling process normally blocks (that is, suspends its execution until the answer is received), unless *asynchronous remote procedure calling* is used, which provides the same functionality as a message-passing system. The remote procedure call becomes the same as sending a message, and the calling process must make another call later to gather the results, which is the same as a receive function.

The standard distribution approach for RPCs, however, is to overcome the blocking nature of remote procedure calls by using multiple processes. In a multi-threaded operating system, *light-weight* processes are inexpensive to start, and share all local memory with their parent except the stack and processor state information. These subprocesses can then be created simply to invoke the blocking RPC, while other processes continue the local execution or other RPCs. In this manner, the distribution is still localised to one machine, and the communication methodology remains simple and predictable.

Although they are easy to understand, and emulate a well-known mechanism for separation of processing tasks, remote procedure calls complicate application development in that they are more delicate than local procedure calls. They can fail in more ways, and as mentioned they do not allow access to global memory structures. Another major limitation when compared with message-passing systems is that they do not provide dynamic communication between tasks, thus preventing processes from co-operating through on-going bilateral communication.

When Sun first introduced RPC, they combined it with a flexible XDR (eXternal Data Representation) package. It allows RPC developers to send various types of data through as parameters without having to do conversions. For example, computers often have different byte-ordering of integer values, and different representation of floating point numbers. An advantage of most RPC systems is that they provide automated methods of data representation over the network.

For some applications, remote procedure calling provides a convenient communication mechanism. Any relatively fine-grained task, with a small amount of data and a small number of parameters, can be distributed in this way. Also, with a multi-threaded process, larger grained tasks can be performed by creating a thread exclusively to initiate and wait for the completion of a remote task.

The Glish [20] system is an asynchronous RPC system implemented at the same level as a shell. It provides data conversion, an interpreted command line and scripting language, and a special library of C++ code used to interface with the scripting language. It also provides a flexible and powerful method of piecing together modules, but it uses a non-standard scripting language, unlike *dish* which is designed to be built in to existing scripting languages.

Glish is not strictly an RPC system, although it looks much like one. A Glish script can start up processes on various hosts dynamically, and then assign operations to *events* (name/value pairs). In this sense, it is similar to Linda [21], except that the operations are defined in terms of processes that the script has initiated. A Linda system must find a process to deal with an object (*tuple*) placed into it. The processes do not interact directly with each other in normal operations.

Another related project is the TCL and Tk systems [13], which provide a generic command line and windowing interface for applications. Tk has a "send" command to invoke TCL scripts on remote machines. However, this is limited by its synchronous nature (like regular RPCs) and also by the windowing system underneath it (X). TCL itself has no interprocess or networking capabilities built in. The distributed shell protocol could be implemented in an extended TCL script language, however.

2.2.4 *Ad Hoc* Distributed Computing

Another category of distributed computing relevant to the *dish* project is making use of the existing but limited distribution tools already available: commands, filters, and protocols such as the *Network File System* (NFS).

When Berkeley introduced TCP/IP to their version of UNIX, they provided a set of user-level commands sometimes referred to as the Berkeley *r-commands*, including `rcp`, `rsh`, `rlogin`, and `rdist`. These commands allow file transfer and synchronous remote execution of shell commands. For example, a programme called `cat` on the local processor can send its output to `sort` on another processor with the following command:

```
cat /etc/passwd | rsh sundog sort
```

Another method is to use the Network File System, a *de facto* file sharing standard for UNIX and TCP/IP. It allows many machines to access the same filesystems or directories. An example of using NFS for *ad hoc* distributed processing is to perform a group of tasks, each creating their own output file. For example, take a set of tests for verifying some software. A shell script can be modified so that it checks for the existence of an output file before starting the related task. If the file already exists, it skips that task and iterates through the remainder until it finds one with no output file created yet. It then executes that task, claiming it by creating the output file. In this way, the same shell script can be invoked on many systems, and the processes will leap-frog over each other solving the problems.

These existing commands and facilities impose many limitations on distributed processing. Execution on each computer must be specified explicitly because there is no built-in mechanism for choosing remote hosts to execute commands. The remote execution protocol is too weak to handle fully interactive commands, but the remote login protocol simulates a full interactive login, thus limiting its use for executing subtasks as needed. This method of remote execution allows no two-way communication, and does not communicate the current *environment* of the user to the new

command. Also, they do not take into account heterogeneous collections of executables. Finally, shells as they are currently implemented, do not allow full use of these facilities for automated shell scripts. For example, job control is often available only for interactive shell sessions and not for shell script execution.

The Expect [22] project provides enhanced distribution via the usual *ad hoc* channels, and is primarily aimed at dealing with automation of interactive processes on UNIX. It uses the pseudoterminal [*pty(4)*] construct to allow processes to talk to each other, locally and over the network. Connections over the network are made transparent to Expect users. Because it was intended as an automation tool for dealing with programmes designed to be used interactively, it provides a novel approach to interprocess communication similar to the approach taken with the *dish* protocol.

2.2.5 Load-Sharing Systems

Load-sharing systems such as *Utopia* [23] provide a general purpose set of services and libraries, similar to the *dish* protocol. Part of the Utopia project was the implementation of `lstcsh` [24], a load-sharing version of the popular UNIX shell `tcsh`. The functionality provided by this shell is similar to that provided by *dish*, but works at a higher level.

The Utopia system is an example of a distributed application building package implemented entirely at the user-level, requiring no change to the operating system kernel. It, therefore, does no process migration. It also assumes that all workstations participating in the system have a uniform filename space (e.g., specifically using NFS to mount all file resources identically on each machine).

The system is based on a distributed *Load Information Service*, which runs on each machine participating in the system. These daemons communicate with each other to keep track of the status of load distribution on the network. In the Utopia design, this service is kept entirely separate from the remote execution service. This provides an advantage of modularity, but a disadvantage of overhead in the process table on each machine and in the network port resources across the network.

Load-sharing systems allow the application builder to tune the application configuration to run optimally on a given network. Unfortunately, monitoring of processor load is not easily done in most versions of UNIX. Often the best measure available is the number of processes waiting on the run queue (known as the *load average*).

Of particular interest is the Utopia load-sharing shell, `lstcsh`, whose units for remote execution consist of entire command lines. The shell maintains lists of commands that are eligible for remote execution and those that are specified to be run locally. It also provides for two modes of operation, one where remote execution of a pipeline is more encouraged, and one where it is discouraged.

Resources are categorised by the needs of each command, and the user can customise resource requirements on the command line. The user can also specify location for execution of any given command. It provides a degree of network transparency for execution of remote commands as if they were local.

Drawbacks of the load-sharing shell are the reliance on the load information server and the overhead of maintaining their record of the network status. Also, commands are not broken up within one pipeline, and a uniform file name space is required.

2.2.6 Determination of Idle Workstations

A problem in any system intended to share workstation resources is how to determine availability of perhaps privately owned workstations sitting on someone's desk. The main difficulty is the unpredictable nature of how people use these resources. Many workstations sit idle most of the time. When users who own workstations want to perform some task, they normally will expect instant response. Also, if they have background tasks of their own, they may wish to disable remote execution of work in order to speed up their own work.

Some systems use an active screen-saver (a process to prevent phosphor burnout on computer monitors by clearing or changing the display after a given time span of inactivity) to determine the idleness of workstations. While this allows users to easily control the remote use of their personal workstation, it causes problems in

situations where background tasks are already running on a workstation that is not in interactive use.

Another common approach is to use the load average available on many UNIX systems. This metric is based on the number of processes in the system's process table that are "runnable," that is, on a queue of processes not waiting for any signal or I/O event. In most cases, any load average of below one is assumed to indicate an idle workstation, but even this can be deceiving in that an editing session will not generate enough cycles to keep its process on the run queue, but having a foreign task start on the workstation may noticeably impair the response of the editor programme, or the graphic display being used.

In addition to determination of idle resources, load-balancing benefits from a flexible method of releasing resources to higher priority tasks. If the system immediately terminates remote execution whenever any activity occurs on the workstation keyboard or mouse, it may cause a process to be terminated prematurely if the user was performing a short task such as checking for incoming mail. However, users may be unnecessarily inconvenienced by a long delay of removing the process from their workstation, if they are resuming work with a task that requires the resources immediately, such as graphic image processing. The owner of the resources, in these cases, should be given control of the availability of their resources to remote users. Most importantly, any restrictions placed on the resource availability should require a time-out factor so that the resources are made available when the workstation controller is no longer physically present.

When a system becomes available or unavailable, a load-sharing system with process migration capabilities can re-arrange the processes as required. This approach incurs a great deal of complexity due to the possibility of open I/O streams. An alternative to process migration is to re-start the task. This requires that there be no error-causing side-effects when the task is repeated on another machine.

The *Condor* system [25] is intended mainly for large computationally intensive single-processor applications. It provides a system where a long process can be shifted

around the network as loads and processor usage changes. It was originally designed as a system for making use of idle workstations on a local area network (LAN). Condor, however, provides no mechanism for interprocess communication.

2.2.7 Graphical Application Builders

The Enterprise [5, 6] and HIGHLAND [4] systems are examples of tools that allow application developers to design and configure message-passing or remote procedure calling programmes graphically.

The Enterprise system allows application developers to write modular code and then piece it together in a graphical environment using a variety of connecting models. It is based on a modified form of the remote procedure call. A developer can tag those procedure calls that would benefit most from being distributed across the network. A special pre-compiler is used to insert the communication primitives into the code and generate the various modules that will be used in the distributed application. Then these modules are *glued* together using a GUI and a number of modules for assigning tasks and gathering results.

Porting sequential code to this system is dependent on the application. It works well for code that is already modular or fits into the paradigm of the system.

The idea of piecing together processing elements in a graphical way could be easily implemented in a graphical desktop using the distributed shell protocol. However, it would be limited by the nature of pipelines and the lack of data conversion. In the context of a graphical shell, however, various tools could be written to piece programmes together, to split up data streams to many processes. These sorts of simple tools could follow the original UNIX philosophy of breaking up tasks to their smallest possible components. For example, a message multiplexor could be implemented by splitting a data stream into lines (strings terminated by a newline character) and feeding each line to a different process running somewhere on the network. Traditional UNIX systems roughly treat files as "databases," lines in files as "records," and delimiting "fields" in lines by white space (tab or space characters) or other charac-

ters. In this manner, records in a database being input could be split and sent to different machines to process.

In the HIGHLAND system, a graphical interface is used to build connections between independently written modules (although they are presented to HIGHLAND as source code). These modules must use specialised procedures to read their standard input and write to their standard output. No other I/O channels are allowed, thus simplifying implementation of the connection process. It uses strongly typed data types and translates them for heterogeneous operation.

The graphical interface that is used to construct the distributed computation remains in effect during execution, thus allowing user control over processes; such as moving them to different hosts and terminating parts of the application.

2.2.8 Process Migration and Checkpointing

Many attempts have been made to provide process migration in a distributed operating environment. These systems typically require extensive changes to the system kernels, and are complex due to the requirement of I/O and other resource continuity. The Condor system [25] allows process migration and checkpointing without system kernel modification, but it uses a system-dependent feature of UNIX: the *longjmp(3)* call.

The essential difficulty with process migration is dealing with processes' active resources, such as open files, sockets, devices, and shared memory objects. The process, after migration, must have the process counter (the processor's idea of where in the programme it is running) set properly. All the information on the stack must also be preserved across the migration, which may require translation if the new processor is of a different architecture, or else the process will not be able to return to lower levels of procedure execution. The code itself may have to be replaced if the processors are not binary compatible. The remaining memory resources being used must also be reconnected. Any open files, communication links, devices, semaphores, message queues, etc. must be recreated or reconnected. In fact, many resources being

used by a given process will be impossible to be migrated to another processor—some process would have to remain to handle that resource usage.

Therefore, either all resources have to be taken over by a distributed operating system kernel (extensively modified from a traditional UNIX kernel), or access to resources must be provided through some centralised service arrangement. In the second case, a serious performance impact can be expected for any I/O or other resource intensive programmes, and applications will only benefit fully from migration with strictly processor bound applications. It also limits the resources that can be taken away from a process to the processor resource itself.

Checkpointing is a simpler approach but requires programmes to be re-written so that they can recover from interrupted computations without the help of the operating system. If a process can periodically write the results or intermediate results of its computation to a file, and use those values to resume computation from the point that they were written out, then it can use checkpointing. In this way, the responsibility for reconnecting resources and preserving the contents of memory are shifted from the operating system level to the user-level. Therefore, the two main advantages of checkpointing are keeping the kernel simpler, and improved performance on a heterogeneous network.

An implementation of checkpointing that works for simple applications on homogeneous networks is to use an operating system call that makes an image of the current executable (commonly provided for debugging purposes) and using it to reconstruct the process on a new machine.

Process migration will eventually be the norm once truly heterogeneous distributed operating systems are available. Meanwhile, substantial gains can be provided by allowing checkpointing if the application developer wishes to use such facilities.

2.2.9 Fully Distributed Operating Systems

Ideally, a truly distributed operating system will provide transparent distribution and migration of processes. Experimental operating systems are being developed to ad-

dress this need. *Plan 9* [26] is a distributed operating system being developed by AT&T, where UNIX was written. Finally, the *Athena* system developed by Massachussetts Institute of Technology (MIT) is currently in use there and provides most of the features of a distributed operating system. Tannenbaum *et al.* [27, 28], at the Free University of Amsterdam have developed the Amoeba distributed operating system.

The distributed operating system being developed at AT&T Bell Labs, Plan 9, uses a UNIX-like hierarchical file name structure to reference all resources in the network. It has a command called `cpu` that allows a shell user to change the processor (Central Processing Unit) being used by the shell, much like the **`system`** command of *lstcsh*. However, users are still aware of the processor boundaries in Plan 9.

Athena can also be viewed as a distributed operating system [29]. This system provides a secure and totally uniform operating environment over a large network. All resources are made available through centralised servers, including authentication and workstation bootstrap software. Processing is done locally however, unless the user explicitly begins a session remote through the Athena windowing system, X. The uniform nature of the network is achieved largely because only two types of workstations are supported. Another probable reason the Athena system has not been widely adopted is because most network installations have many resources that are already in use and system administrators are unwilling to give them up to the Athena system, since it provides security by taking over control of the workstations.

In Amoeba, the notions of processes and files are replaced by the more general *object* (a collection of data), which may be active, or dormant in some storage area. For example, a user login session is simply a data object. When the user connects to a workstation, the object is brought out of long term storage (most likely a hard disc drive) and into a processor's memory. Upon disconnection the object is migrated back out to disc. *Capabilities* are used to name objects and to perform services on them.

Communication in Amoeba is limited to remote procedure calls, implemented as

three distinct system calls. Processes must locate the service they desire through the hierarchical naming service, and make a system call to invoke the remote procedure call. Servers must provide their service name to the operating system through another system call, which also provides them with an incoming request. The server terminates the remote procedure call with a third system call. Amoeba also supports multi-threaded programming.

A main goal of Amoeba was to remove any centralised reliance on processors. It provides a pool of processors beyond those contained in the workstations. Amoeba users do not know how many processors they are using at any given moment nor where they are physically located. Processors in the system run a kernel that provides minimal communication and core services. Amoeba has a UNIX emulation mode, although no UNIX code was used in its implementation.

2.2.10 Security

Security in distributed systems consists of two procedures, *authentication* and *encryption*. Authentication ensures that agents on the network are whom they claim to be. Authenticated agents can then be authorised to do certain tasks, or be held accountable for performing questionable tasks. Privacy, or an encryption system, is sometimes required in sensitive applications and must be provided in any useful distributed system.

There are three commonly used authentication methods. The first and most cumbersome is requiring a password every time a connection is made over the network to another node. This is the most secure since each machine can keep its own list of passwords and authenticate each user every time. However, it is highly inconvenient and a shell user would rapidly tire of having to type passwords to the different machines on the network. The slow response time would also preclude any kind of rapid deployment of work over the network. Although effective at securing the system from external networks, this method generates traffic on the local area network containing passwords that could be used to compromise the system.

Another approach is that employed by the Berkeley *r-commands*. The servers that respond to these command requests check two files to see if they should grant the request [*rlogind(8)*]. One is a system wide file (`/etc/hosts.equiv`), and another is a file in the home directory of the apparent requester (`.rhosts`). This system allows authentication to be performed locally on the server machine without checking passwords, and requires that these files be carefully protected on each system in the network. This is the least secure system since it relies only on the IP address of the requesting machine to match its tables, and it might be possible under some circumstances to subvert this check by impersonating a machine on the local area network that is currently down.

The third approach is to use *Kerberos*, the authentication server developed for the Athena system at MIT [30]. Kerberos provides *tickets*, time-limited access keys that can be used for authentication to services, to its clients. Tickets are distributed by a centralised, carefully controlled and secured server. To make the Kerberos system secure, the clocks on all nodes must be synchronised [31], although other protocols have been designed to overcome this limitation [32].

The other security measure, encryption, is typically done with a *private key* system. Both parties need to have a secret piece of information that can be used as a key for an encryption algorithm. Given a piece of text and a key, this type of encryption algorithm *encrypts* the text using the key, resulting in unreadable text output. This output, when fed back as input to the encryption algorithm with the same key, produces the original text. Private keys for this purpose can be obtained from protected files (in a file readable only by the invoking user), or else can be generated by the Kerberos system.

An alternative to the key broker is to use *public keys* [33]. These keys consist of pairs, one half is kept secret to the client, and the other half made public. Unlike the private key systems, a key will not decrypt a message encrypted with itself. Text encrypted with one half of the pair can only be decrypted with the other half. Therefore, one agent can send a message encrypted with the public key for its desired

destination and only that destination node can decode it.

2.3 Distributing the Shell

In a fully distributed operating system, the shell or user desktop would automatically use resources transparently across the network. However, the primary paradigms for distribution of processing currently in use do not address such high-level requirements. Socket programming, message-passing libraries, and multi-threaded remote procedure calling all operate at a level too low to be useful from the shell level. Other protocols, such as load-sharing systems or scripting languages such as Glish, are more generally useful, but not tailored exclusively to provide maximum performance at the shell level. The load-sharing `lstcsh` shell provides some functionality of a distributed shell, but is limited by the Utopia environment, which was designed for more general purpose applications. Process migration (if supported by the underlying operating system) and checkpointing can be provided more simply if limited to the shell construct. Distributed shells based on GUIs would borrow concepts from the graphical distribution tools such as Enterprise and HIGHLAND. The primary advantage of specifically distributing the shell is, however, that it perpetuates the original philosophy of the UNIX system—making the shell responsible for piecing together many smaller pieces in order to provide a flexible and powerful computing environment.

Chapter 3

Design of the Distributed Shell Protocol

A complete description of the distributed shell protocol is presented here. It can be used to provide distribution of task execution for shells, graphical desktops, and other general-purpose scripting languages such as Perl or TCL.

Dish is designed in the client-server paradigm, which is the primary model for distributed applications on TCP/IP networks. In the *dish* system, a *server* runs on each participating computer. Each server provides task execution for many clients. An interactive session with a user is a *client*, as well as any executing *dish* script.

An analogy is used to describe the allocation of tasks. Servers are likened to potential employees or contractors, and clients to employers. When a client has a subtask to do, it *advertises* it over the network. Servers respond with *applications*. The client then chooses the most appropriate application, and *offers* the job. Figure 1 shows the relationship between servers and clients.

3.1 Design Considerations and Constraints

The following considerations went into the design of the distributed shell protocol.

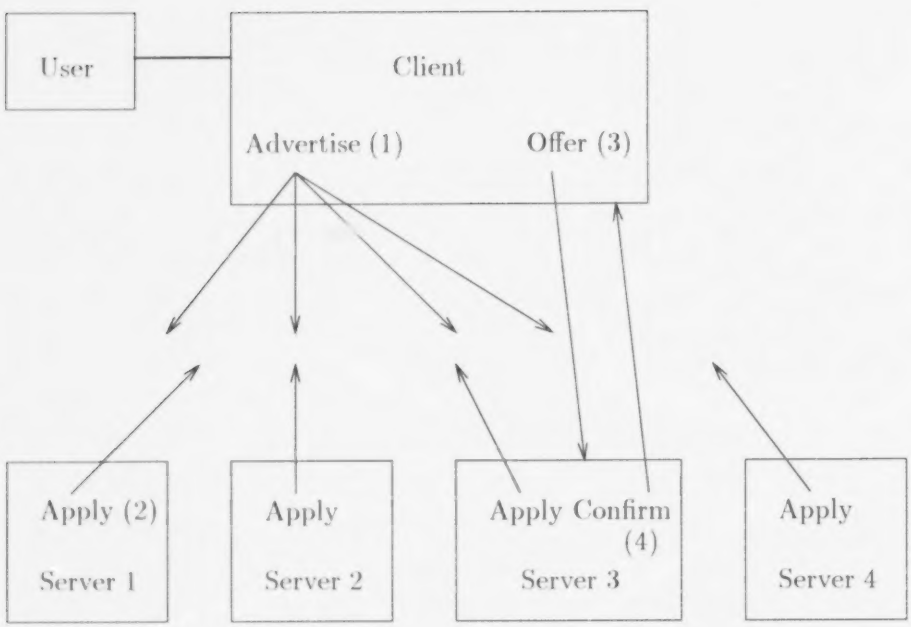

Figure 1: Client and Server Relationship

- The protocol should be independent of any particular shell, command line interface, or graphical desktop. It should be generalised to make it useful for any system with multiuser, multiprocessing capabilities.
- No database of load information should be kept. Each time a subtask is executed remotely, the network should be queried and response time can then be used as the main heuristic for the shell to choose a server.
- The functionality should lie between the existing remote execution protocols (the Berkeley '*r-commands*') and higher level tools such as Sun RPC and Enterprise.
- It should bridge the gap between current client/server technology and the functionality of a completely distributed operating system.
- It should follow the distributed process paradigm of making small modules that need not be aware of the details of their connection to other modules.
- The protocol should be easily extended in order to allow fully transparent remote execution of tasks, including terminal control.
- The protocol should allow for robust network error recovery, and for socially acceptable use of available resources.
- Process checkpointing services should be provided.
- Support for interactive programmes such as screen editors should be provided.
- The protocol should include the ability to extend the traditional notion of a UNIX shell pipeline including splitting and looping.
- The shell command line or desktop interface should be extended to allow the user to control at least some aspects of execution over the network.

In addition to these design considerations, the protocol was designed with the following assumptions:

- The protocol is designed to fit with the set of protocols built on top of the Internet Protocol (IP). Another protocol at the same level is the *rexec* protocol. Protocols of this type are documented in RFC 1340 [7], and are usually listed in the UNIX file `/etc/services` [*services(5)*]. The programming interface used is the Berkeley *socket* library package, the original implementation of TCP/IP on UNIX.

- The underlying network provides fast datagrams and slower stream connections. This is the case for most UNIX workstations running TCP/IP protocols.

- Network bandwidth and processors will become cheap. In the near future large numbers of processors will be available for remote execution. This scenario favours a design that relies on small, independent servers able to respond quickly on a fast network over a design that favours large centralised databases or servers running on mainframes.

3.2 Interaction of Client and Servers

In the distributed shell environment, there are two main classes of interprocess communication. The first is the control messaging between the *dish* server and client processes. The second class is interprocess communication that takes place between the subtask processes spread around the network. The different types of communication channels are shown in Figure 2. Small boxes represent processes and the large boxes their host computers. Above the dotted line are the client and server processes, which send each other command messages with UDP datagrams, shown with dashed arrows. Below the dotted line are the subtasks, which initiate their own TCP Interprocess Communication (IPC) stream connections, shown with solid arrows indicating the direction of data flow. Subtasks are children of the processes shown above them, and can communicate using whatever process is most convenient, usually pipes [*pipe(2)*].

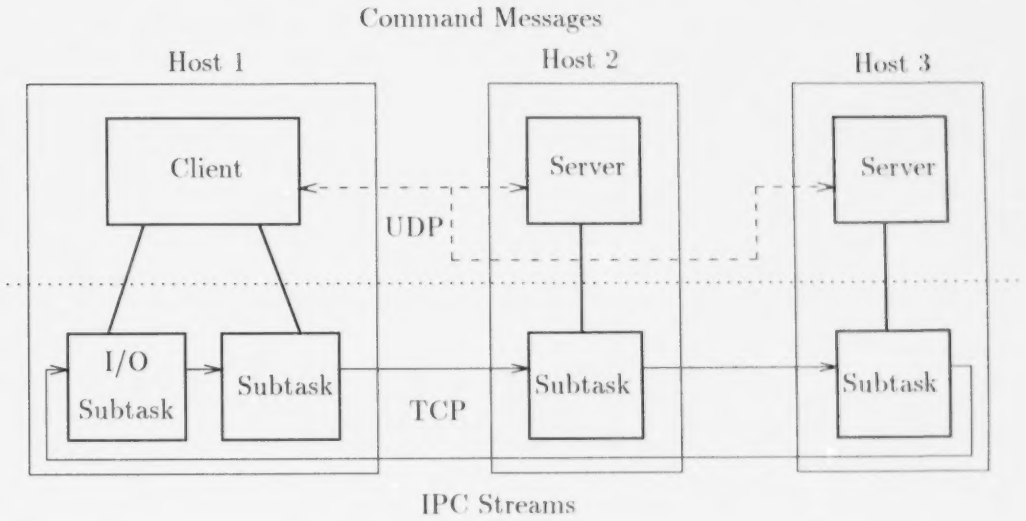

Figure 2: Interprocess Communication Channels

A client and its server processes communicate via datagram packets to provide control over the system. Datagrams are used to make the initial process of task start-up and negotiation as fast as possible. UDP as defined in the Internet Protocol Suite is quick but not guaranteed or reliable. For the distributed shell however, unreliability is not a problem because if the client encounters an error during start-up, it will not want to execute the subtask on the associated server anyway. Absence of a response is quietly ignored by both server and client.

However, to get this quick response, the *dish* protocol must include the small amount of reliability and flexibility that a higher level (but slower) protocol such as TCP would provide. This is done by including a task identification field in every message, by providing a mechanism to break up large messages (for example, sending a large environment description to a task about to start execution), and by providing user-adjustable timeout factors, after which servers and clients will decide that the other party has failed and will give up on it, taking whatever action is required upon failure.

Advertisement	0
Application	1
Offer	2
Confirmation	3
Completion	4
Checkpoint	5
Checkpoint acknowledgement	6
Quit	7
Signal	8
Signal acknowledgement	9

Table 3.1: Dish Control Message Types

Subtasks of the server and client processes use the information contained in these control messages to connect themselves with TCP streams. These connections, once the subtasks are started, appear to be their standard input, output, and error channels.

Ten possible *dish* control message types, and their values for identification in messages, are shown in Table 3.1. Each command message type has required fields and optional fields, depending on the information needed in different circumstances.

Servers listen for these messages on an agreed upon port number. The Network Information Centre provides a port number registration service [7]. Currently the *dish* protocol has no registered port number, so a port number must be decided upon by the installation procedure in order not to conflict with other unregistered software protocols installed on any given system.

Servers may take requests from any number of clients at this one port number and must keep track of their various clients internally with the help of the information contained in the messages.

The message format itself is defined as follows. The message consists of regular ASCII-encoded characters (the *American Standard Code for Information Interchange* provides the standard mapping from characters to binary storage on UNIX computers). They are divided into fields each separated by a newline character. Each field

consists of an identifier, followed by an '=' sign, followed by the value assigned to that field. The first line must contain a version field, which for this definition of the *dish* protocol is equal to zero. Therefore, the first five characters of all *dish* command messages are "`dv=0\n`," where "`\n`" is a newline character. The remaining fields may be in any order (except for the `env` parameter described in Section 3.2.3). Their names and permissible values are described in the following sections. Some fields are common to multiple message types, and any extraneous fields are silently ignored. A summary of all parameters is given in Appendix A.

3.2.1 Advertisements

A client shell wishing to perform a pipeline over the network first broadcasts an advertisement for each piece of work making up the pipeline. It is the responsibility of the shell implementation to determine whether such advertisements are required for each part of the pipeline. For example, some commands might be marked as locally executable only, or they might be built-in to the shell.

The job advertisement is a UDP datagram sent to a default or specified list of *dish* server hosts. UDP datagrams are used to make the job application process as fast as possible. The shell's method of tuning the timing parameter helps prevent busy stations from getting more jobs than they can handle. Any *dish* server can reply to a job advertisement, including one on the same machine. This enables the system to take advantage of multi-processor machines by allowing a *dish* server daemon to negotiate jobs on the different processors.

Table 3.2 shows the fields available for job advertisement messages. Optional parameters are given in *italics*. The `dv` parameter is the *dish* version parameter. Currently it is always set to "`dv=0`." The `type` parameter indicates what kind of command message is being sent. Advertisements are defined to be type `0`. Table 3.1 shows the other values available for the `type` parameter. Appendix B contains an example of an advertisement message.

The `user` parameter specifies the client's user name. The *dish* server authenticates

`dv`	*cwd*
`type`	*umask*
`user`	*ticket*
`job_id`	*host*
`client_port`	*path*
`command`	*arch*
	os
	memsize
	cpuperf

Table 3.2: Advertisement Message Fields

the client's identity by checking the file `/etc/hosts.equiv` or by reading the file `.rhosts` in the user's home directory, obtained by looking up the user name in the file `/etc/passwd`. Alternatively, the `ticket` parameter can be used with a third-party authentication server such as that provided by the Kerberos system from MIT [30].

The `job_id` parameter is a unique string to identify a job. It consists of three parts, the host name of the client, the client's process identification (PID) number [*getpid(2)*], and a sequential number, increased by one for each new job advertisement. This string identifies the job in question for the remainder of the client's lifetime, so it must be designed to be unique for any possible client.

The `client_port` parameter specifies an Internet Protocol port number for the server to send applications to. The parameter `host` for this message type indicates the host name of the client in case it is different from the sender of the advertisement—if the client is using a *broadcast server* for the job applications on a different node on the network, then the sender address will not be valid for returning applications.

The `command` parameter contains the name of the executable file that the client wants performed. There are three possible ways to reference a command.

- An absolute pathname may be given. In this case an executable file is described with all directories starting at the root (e.g., `/usr/ucb/vi`). If a *dish* server receives such an advertisement, it can immediately check that one location to see if there is a valid executable by such a name and respond immediately.

- On the other hand, a relative pathname may be specified. This is a pathname with directory names but not starting at the root (e.g., `./mybin/myprog`). Then the `cwd` must be provided in the job advertisement, so the server can check immediately if such a file exists in the server's file space. This can fail also if the current working directory does not exist or is unreadable on the server node.

- Finally, a simple command can be specified, which consists of a simple name with no directory specifications. In this case, the `path` must be specified in the job advertisement.

A traditional shell searches the path for the specified file name and attempts to execute it, regardless of its permission bits and magic number (see Section 2.1). The distributed shell has to ensure that it can in fact be executed on the server's hardware. It may be unable to do so because the executable itself is intended for a different processor. Also, an executable script may be written for a shell that does not exist on the server's platform. For example, if it is a script specifically written for Perl and has the first line of `#!/usr/bin/perl` and the server has no version of `perl` available in `/usr/bin`, then it cannot reply with a job application. Similarly, the client itself has to build its hash table containing only executables it can perform. This added limitation will have an impact on the performance of a distributed shell, because the traditional shell has the advantage of being able to assume it can execute everything and then fail when it makes the attempt.

The `path` parameter contains a set of directory names, separated by colons (':'). The server uses this list to attempt to find an executable file if the `command` parameter has no path. The value of the `path` parameter is taken from the client's `PATH` environment variable.

If the path includes the current working directory (specified by a "." as a member of the path set), or a relative pathname, then the `cwd` field must be provided in a job advertisement as well. In this case, the server subtask invokes the *chdir(2)* system call before doing the executable search, so that the current directory (indicated by a

.) is valid. The current working directory must always be specified as an absolute pathname.

Two parameters, `cwd` and `umask`, contain the current working directory of the client and its file creation mask [*umask(2)*], respectively. These are two parts of the shell environment that potentially change frequently. If they are not specified, the user's home directory is used as the current working directory, and the system default umask is used for the file creation mask. Operating systems other than UNIX will have different parameters of this type specific to their operation.

Other parameters describe requirements of the process. These will come from a table in the *dish* client, set up by the user to tune performance of the *dish* application. To specify a processor type the `arch` parameter is used. The format is the output of the `uname -m` UNIX command [*uname(2)*]. This parameter can be used for compilers and other programmes where the output will be different depending on the processor architecture. If a specific operating system is required, it can be identified with the `os` parameter, the value of which is the output of the UNIX command `uname`. Version numbers can be appended to this field's value to provide more specific operating system requests.

Resource specifications currently include memory size and processor performance. An estimate of memory usage is provided by `memsize`, and processor requirements are estimated by `cpuperf`. Memory is measured in megabytes of main storage. Processor performance can be a function of a static processor speed index and the current load on the system, or for more sophisticated load-balancing, a dynamic benchmark could be run each time a job application is processed. Unfortunately, however, a dynamic benchmark would slow the application process.

On a fast network capable of broadcasting to all stations on the network, a client could use a *broadcast server* to send out job advertisements instead of using a list of servers. The distributed shell client cannot send the broadcast itself, since in the UNIX environment using the underlying broadcast capabilities is a privileged operation. Therefore, if the `host` field is specified in the job advertisement, servers

`dv`	*memsize*
`type`	*vmemsize*
`job_id`	*cpuperf*
`host`	*processors*

Table 3.3: Application Message Fields

must use its value to look up the IP address of the client. Otherwise, the server can more speedily extract the address from the advertisement message [*recvfrom(2)*].

3.2.2 Job Application

A *dish* server wakes up when it receives a job advertisement broadcast. It checks to see if the system is idle, then decides whether it qualifies for the job, and if it does, sends a job application to the potential "employer." The address of the employer is constructed from the port number specified in the advertisement message, and an IP address found with the `host` field [*gethostbyname(3)*] if present, or else taken from the sender address of the advertisement. Fields used in application message are shown in Table 3.3.

Again, the `dv` field contains the version number of `0`. The `type` for an application is defined as `1`. The `job_id` contains the same job id string as presented in the job advertisement. All future messages pertaining to this job will contain an identical string for this field.

The `host` parameter specifies which host is applying. Currently this is a required field, but in future a default could be derived from the sender's address as is implemented for advertisement messages.

The remaining parameters, if present, provide the client with more information to use for the server selection. The parameters `cpuperf` and `memsize` represent the processing capabilities and physical memory size, respectively. The parameter `vmemsize` is the size of the server's the virtual memory space. If a server has more than one processor, the `processors` field should be provided. The value is an integer indicating

the number of processors available in the system.

When a server has sent a job application, it returns to its original state, awaiting other advertisements or offers.

3.2.3 Selection and Job Offer

Distributed shell clients listen for job applications for a specified period, called the *application timeout*. The optimal value for this timeout is a tradeoff between network performance and response time for the user. For interactive shells in particular, the timeout must be short to make the delay unapparent to the user.

If a user has a shell script that invokes resource intensive subtasks, it may be worthwhile to include in the script some tuning adjustments in order to give the client more time to find the best server available on the network.

If no applications are received before the timeout arrives, or if the client decides that its own host is better qualified than those of all received applications, the client attempts to run the command locally. Any applications arriving late will be silently discarded.

Once the client has chosen a server, it composes an *offer* message. This confirmation will include all the detailed information required to run the command appropriately, such as the environment, and instructions on how to connect current file descriptors. *Offer* parameters are shown in Table 3.4. The message `type` for offers is 2. The `dv` and `job_id` fields are as described in section 3.2.1.

The number of arguments to the command is specified in `argc`, and the arguments themselves are specified in `argv`. The arguments are listed in order, contained in square braces (`[` and `]`) to prevent the ambiguity of quoting. By convention, the command name itself is included as the first argument.

The two parameters `ports` and `hosts` provide the necessary information for the servers to set up the required pipelines between processes. On a UNIX system, each command usually expects to have three files open when it is invoked, the standard input, standard output, and standard error channels. Some shells, such as the Korn

`dv`	*env_count*
`type`	*encryption*
`job_id`	*protocol*
`argc`	
`argv`	
`ports`	
`hosts`	
`env`	

Table 3.4: Offer Message Fields

shell [*ksh(1)*] allow other files to be opened on behalf of subprocesses by the shell. Therefore, for each I/O channel needing to be opened and actively connected (see Section 2.2.1) for the remote execution, a host name and port number are given. Hosts are given by name, separated by colons (:), and the corresponding port numbers are simply decimal integers also separated by colons. If a host and port number is missing for a channel in the offer, it means that the server subtask is expected to wait passively for an incoming connection request, and it must provide a port number for the other subtask to use to connect to the passive socket in its *confirmation* message. The server must then find and allocate an available port number and send it back to the client. If the host name and port number are given for a channel, the server is expected to make an active connection to the port at that host's address.

A third optional array of values specifies which protocol to use [*getprotobyname(3)*] for each interprocess channel instead of the default TCP, and is called `protocols`. Maximum performance can be achieved with reliable networks by using lower protocol levels. A special value for the protocol field is `pty`. This invokes the Berkeley *rlogin* protocol over a TCP line corresponding to the standard input, or the output and error channels [34]. This protocol uses a pseudoterminal in order to make the specified channel appear as a terminal to the subtask.

The `env` parameter contains the environment space for the job. This parameter must be last because it might contain newline characters. If the environment does not fit into one UDP message, it will also be continued in the next `env_count` messages

dv
type
job_id
host
ports

Table 3.5: Confirmation Message Fields

to follow. If the messages are not received immediately, the remote execution fails. Subsequent offer messages then have a decrementing value for `env_count`.

The `encryption` parameter allows clients and servers to use a data encryption scheme. This field will include the scheme name and the key if required.

The offer is sent to the potential server and the shell client waits for a job confirmation message.

3.2.4 Job Confirmation

The final step in the four-stage startup is the confirmation from the *dish* server to the advertising client. The confirmation message also serves the function of returning the port numbers to the client upon which the server will wait for incoming interprocess channel connections.

Table 3.5 shows the parameters required for the confirmation message. Before sending its confirmation the server must find available port numbers for each passive socket it needs to open, and send these values back to the client through the parameter `ports`, thus allowing the client to send the port number to the task on the other side of the connection.

3.2.5 Job Completion

When a server detects completion of a subtask, it must send a final message to its client. Table 3.6 contains the parameters for a completion message. The `type` parameter is `4`. The return code from the subshell execution is returned in the `status`

dv
type
job_id
status

Table 3.6: Completion Message Fields

parameter.

The five message types described are the core of the remote subtask control facilities. The remaining message types are described in the following sections, as are the semantics of setting up the pipelines between subtasks. The approach taken is decentralised, using the network and server reaction performance to job advertisements as part of the mechanism for choosing the most appropriate server to perform a task. It can also adapt more easily to changes in the network, since no state is assumed when a job advertisement is broadcast, making it more reliable than systems that have a static allocation of resources. Unless the application can only run on remote servers, a workstation taken entirely off the network could still function by executing all commands locally. In comparison, *Utopia*'s load-sharing module performs a similar function but constantly generates traffic on the network, even when the load sharing shell is not in use. The *Glish* system requires the user to specify the name of the server for any given task, and therefore provides no mechanism for automatically distributing tasks.

3.3 Pipeline Setup

The next step once a task has been confirmed by a client for a given server is to set up the pipeline channels. If the task is the first in a pipeline, the client shell is responsible for providing input on a socket of the appropriate protocol type (see the `protocol` parameter in Section 3.2.3). If the task is the last in a pipeline, the client must arrange to pick up the standard output and error channels in the same way, and

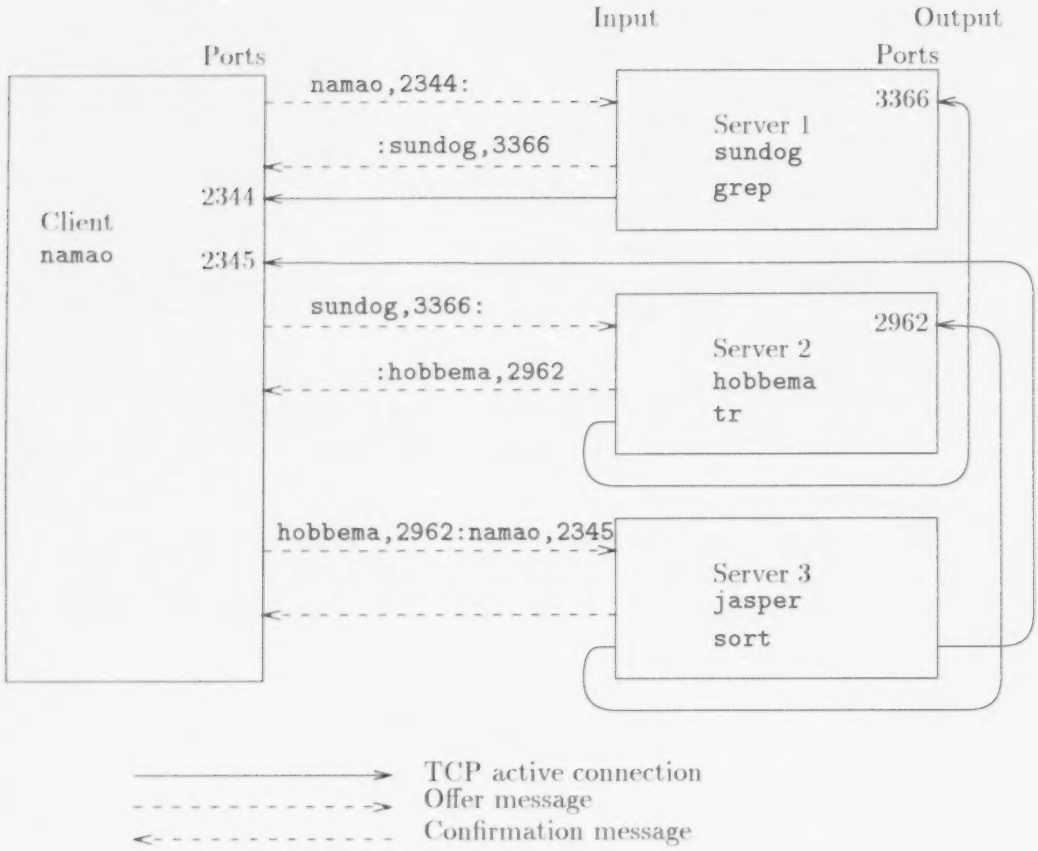

Figure 3: Pipeline Setup

display them to the client's user. A single task can require both input and output if it is the only member of a pipeline, and if a shell is being implemented that allows other file descriptors to be open for a subprogramme (e.g., the Korn shell or Perl), they are set up in the same way.

These connections can be set up as TCP or UDP connections. The Transmission Control Protocol is generally used on unreliable networks where packets can be lost, duplicated, or delivered in a different order than they were sent. Unlike the sockets used in the initial task set up, which are not connected for purposes of maximising response time, connections are required for the actual inter-task sockets. Since the

dish sockets are set up to provide two-way communication, applications can take advantage of this to send information both ways on the pipe.

The Berkeley socket paradigm requires a *passive* and an *active* side for socket connection. The client is responsible for deciding which side of a connection is active and which is passive, and shows this through the parameters in its job offer message. This is the client's responsibility because it must gather port numbers from each server before it can send them on to the next server. The input to the first task and the output from the last task are automatically assigned to the client task itself. The client thus chooses port numbers and fills in the appropriate offer messages to the first or the last client. The server does not have to know or care whether the sockets it is opening are going to be for input or output, as long as the client ensures that one side of each connection is indicated as passive and the other is indicated as active, by the presense or absence of a corresponding field in the `hosts` and `ports` parameters of the job confirmation message.

Figure 3 shows an example of how a client might set up the active and passive socket addresses for a pipeline executing on three different servers. The command issued to the client *dish* on host `namao` produces all the words in the system dictionary ending in the letter 'y,' converted to upper case, and sorted in reverse order:

```
grep 'y$' /usr/dict/words | tr '[a-z]' '[A-Z]' | sort -r
```

The host `namao` first advertises the three jobs in the pipeline. All applications received in time are processed, and the three commands are assigned to servers.

In this example, an offer is sent to `sundog` to perform the `grep` command. Included is the port number (2344) on which `namao` will supply the standard input stream (although in this case it will not be used). `Sundog` replies with a confirmation message, including the port number (3366) for the socket on which it will send its output. After this exchange, `sundog` will make an active connection to `namao` at port 2344, and will passively wait for a connection on its port number 3366.

The port number returned by `sundog` is then included in the offer message sent by the client to `hobbema`. The remaining port numbers are communicated similarly.

Note that the arrowheads in the TCP connections point to the passive side of the socket connection. Input and output functions are indicated by the side of the box on which the connection is shown. The direction of communication on a TCP connection is not related to which side is active or passive.

If a server is the successful applicant for a job and then fails before sending a confirmation, the client must either take over that subtask or choose another server. For this reason, if a server subtask fails to make the connection properly in a given amount of time, it must assume that one of its neighbours has failed, close down all its sockets, and try the process again. After a pre-specified number of tries the server subtask gives up and the pipeline fails. Server failure after transmission of confirmation messages is also considered a general failure.

Once the inter-task communication pipelines are connected between subprocesses on both the client and server side, the client has two ways of determining failure of the connection. First, a signal message may be sent from the server when it notices its subtask terminating abnormally. Second, if both the server and its subtask fail and disappear, the client will notice that no checkpoint messages (see Section 2.2.8) have arrived for a specified amount of time and assume the machine has failed. Then it must close down its subtasks and start them again to prevent the server from coming back on the network and resuming its work. If a server detects the failure of its subtask, it can send a signal to the client with a *signal* message. The server is also responsible for transmitting signals to its subtasks.

3.4 Signals and Job Control

Normally when a shell executes a subcommand locally, the subtask is responsible for handling its own signals. Shells need to deal specially with certain signals regarding I/O stages and interrupts. For the distributed shell, the communication has to be negotiated through the servers since the subtasks may be executing somewhere else on the network. Therefore, *signal* messages can be sent from client to server or server

dv
type
job_id
signal_id
signal

Table 3.7: Signal and Acknowledgement Message Fields

to client to indicate the reception of a signal by the subtask or to send a signal to a subtask. An example for an implementation of *dish* with job control, would be the `SIGSTOP` and `SIGCONT` signals to stop and restart a job, respectively.

To provide network transparent signal delivery, servers must install interrupt handlers for all relevant signals before forking [*fork(2)*] and invoking the subtask. This allows the server to inform the client shell of any unusual activity taking place on the server machine.

A three-way handshake is used to ensure that the signal is received. First a *signal* message is sent to the server or client process. It then communicates the signal to the appropriate process on the server or client node. Once the signal has been successfully delivered, the signal is acknowledged through another control message. Finally, the acknowledgement is also acknowledged.

3.4.1 Signal Messages

Table 3.7 shows the parameters required in signal (type `8`) and acknowledgement (type `9`) messages. Acknowledgements do not need the `job_id` nor the `signal` parameters. If the control message is being sent by a client to a server, the `job_id` parameter specifies which subtask is to receive the signal. Otherwise, if the server is sending the *signal* message, it shows which subtask generated the signal. The `signal_id` parameter is a unique string used to connect acknowledgements to their signals. It consists of the value of the `job_id` parameter, concatenated with an '`s`' (for server message) or '`c`' (client), and another sequential counter. Finally, the signal value itself is contained in the parameter `signal`, and consists of a POSIX signal number [*signal(3)*]. (POSIX

is a set of standards for UNIX systems, including a standard signal interface.)

Interprocess signals are the most operating system dependent part of the *dish* protocol. To provide a more generic protocol, future versions of the *dish* protocol will need a more versatile signal sub-protocol.

Note that the *quit* control message replaces the functionality of the termination signals and should be used instead of those signals.

3.5 Security

When a server receives a job advertisement, the associated user is authenticated by looking for the client's machine name in the files `/etc/hosts.equiv`, and `.rhosts` in the user's home directory. This scheme, taken from the Berkeley implementation of the *r-commands* allows the user to set up access privileges without intervention of the system administrator, and trusted local area networks can take advantage of their internal security [34].

This security mechanism is convenient because it allows users to access other systems without providing passwords. However, there is a large security weakness on LANs using this scheme because a malicious user or an intruder can set up a workstation masquerading as another temporarily disabled workstation. It also allows access to the entire LAN when one machine is compromised.

For extra security, the distributed shell servers can use a private key broker such as Kerberos [30] to authenticate each client, and provide private keys for the processes to use for encryption purposes within the *dish* system. A public-key encryption system could also be incorporated into a distributed shell for specialised applications requiring strong encryption and authentication.

To achieve the goals of flexible security, the *dish* protocol is designed to isolate the authentication and encryption functions so they can be easily tailored to the application's need.

3.6 Control and Error Recovery

The *signal* and *quit* command messages are used for process control and error recovery. The *checkpoint* command message is used to help in error detection.

Two situations arise where a distributed shell may have to backtrack—if a certain station fails, or if another user on that station requires the resources being consumed by the remote task.

If a server notices a subtask failure, it is responsible to report it back to the client via the *signal* message mechanism. If the server itself fails, the client will discover this by the lack of checkpoint messages. To detect the loss of a communication channel for TCP connections a `SIGPIPE` signal would be received, then the *dish* server is notified and a new process is started. A limit on the number of restart attempts would be wise to include in a *dish* implementation.

In the case of workstation users, who have a computer on their desk that is not always busy, they must be able to communicate with the *dish* server and tell it to stop doing work or to resume work. There are many ways to implement such a feature, depending on the local circumstances. For example, a convention could be adopted that the existence of a file called `/tmp/..no_dish..` indicates that no remote tasks should be executed on the local station. If the temporary directory `/tmp` is not local to the workstation, then the hostname would have to be added to the filename in question. Traditionally on a UNIX system anyone can create or delete files in `/tmp`, and the directory is cleaned of old files regularly. However, the mechanism to disable the remote *dish* tasks must have enough privilege to send termination signals to the resident *dish* subtasks. It would therefore be best built in to the server process.

A UNIX command or X tool could be provided to disable or enable remote *dish* tasks, and to terminate any processes already residing on the system. Alternatively, a programme to lock the screen of a workstation could be built that toggles the activity of the *dish* server, so that whenever the user locks the screen of the workstation to leave for a while, the station would be made available to remote tasks. Depending on the circumstances of the installation, it may be desirable to introduce a delay between

dv
type
job_id

Table 3.8: Quit Message Fields

disabling new *dish* subtasks and destroying the old ones, thus allowing short tasks to complete without having to restart somewhere else, but still eliminating long tasks from interfering with the interactive performance of the workstation.

3.6.1 The Quit Message

The quit message has parameters as shown in Table 3.8, and is of `type=7`. A client sends this command message to a *dish* server if it wishes the server to terminate the subtask shown by `job_id`. A server does not send this message type to the client. If a server detects a subtask has abnormally terminated, it sends an appropriate *signal* message, waits for the appropriate acknowledgement handshake before shutting down the subtask.

3.7 Checkpointing Service

The checkpoint messages serve two distinct functions. The first function, as the name implies, is for servers to send checkpoints of the progress of their subtasks to their client. The second function is to provide detection of errors that are not caught by the regular socket interface routines.

If a client fails to receive a checkpoint message from a server within a user-configurable time factor, it assumes the server has failed and fails itself, or restarts the process elsewhere. If a server fails to receive an acknowledgement from a client within another user-configurable timeout factor, it assumes the client has failed and terminates the client's subtasks.

To make full use of checkpointing, programmers must change their code. First,

they must make it so that upon entry, their programme checks its environment for any variables another instantiation has placed there for it with partial results or instructions to resume a partially completed task. Second, shell scripts must be modified to allow or to disallow checkpointing, since many existing programmes will be incompatible with the checkpoint approach. If full checkpointing is not specifically enabled, the checkpoint messages will still be sent back to the client, but no environment space will be included.

3.7.1 Implementation

The checkpoint service complicates programme implementation because it requires that the server have some way of polling the environment space of its subtasks. This can most easily be done with a named pipe [*mkdev(2)*], and a user-definable signal (or a new signal if kernel changes are deemed appropriate or desirable), which the server sends to its subtasks in order to have them place their current environment space onto the pipe for the server to read and send on to the *dish* client.

On the client side, when full checkpointing is in effect, all communication between subtasks must be stored up between checkpoints. This requires all socket connections to go through the client processor instead of being directly connected to subtasks, thus slowing communication performance. Therefore, a checkpoint acknowledgement message may include a parameter to adjust timing in case the client buffers become full.

Since programmes have to use the *dish* method of storing checkpoint information in the environment space, checkpointing must be enabled for any session or script that wishes to use full checkpointing facilities.

3.7.2 Checkpoint Messages

Parameters defined for checkpoint messages are given in Table 3.9. The subtask for which the checkpoint is being made is described by the `job_id` parameter. The checkpoint message from the server to the client includes the `env` (and optional

`dv`	*env*
`type`	*env_count*
`job_id`	
`timestamp`	

Table 3.9: Checkpoint Message Fields

`env_count` for environments too large to fit in one datagram) parameter described in Section 3.2.3. Checkpoint messages from the server to the client require a new, unique timestamp, which is a string representation of the number of seconds and milliseconds since the UNIX epoch [*gettimeofday(2)*], separated by a colon (`:`), as measured on the server's clock. The same `timestamp` is used in the checkpoint confirmation message as returned by the client back to the server.

To make use of the *dish* checkpointing facility (see Section 3.7), an application must place information in the *environment space*, and check for that information when starting execution to resume execution where it last sent the environment space back to the client.

3.7.3 Checkpoint Acknowledgement Messages

The *checkpoint acknowledgement* message (Table 3.10) is provided in order to speed up or slow down the checkpoint messages, and also to let the server know the client is still running. The `timestamp` parameter is the timestamp of the checkpoint being acknowledged. The `interval` parameter has the same format as the `timestamp` parameter, showing the desired amount of time to elapse between checkpoint messages. This message is only sent from the client to *dish* servers, and is provided because with heavy I/O usage the client buffers may overflow and cause excessive blocking of subtasks.

dv	*interval*
type	
job_id	
timestamp	

Table 3.10: Checkpoint Acknowledgement Message Fields

3.8 Processor and Resource Classes

The *dish* servers are responsible for determining whether they can execute a given command by searching their path for an executable file valid for their own processor type. This is done by the *dish* server extracting the path parameter in the job advertisement and searching for a valid executable in that path. In this manner, no database of processor classes is needed. Furthermore, no polling over the network is needed to keep track of resource availability at any given time. Most other load-sharing and distributed task allocation systems require either a centralised or distributed database of capabilities that must be kept up to date using queries over the network. This causes extra traffic on the network and a potential centralised bottleneck. These problems are avoided by the four-step *dish* startup mechanism, which reflects resource and processor availability only when needed.

Processor types can be specified if required, for example, in the case of a compiler where a certain type of executable is desired. Then the client simply includes the `arch` parameter in its job advertisement. Furthermore, if the *dish* client is told that a certain programme is processor intensive or memory intensive, it can also specify those requirements in the job advertisement.

The following resource categories are defined: processor architecture, real memory size, virtual memory size, and operating system. A more elaborate description of storage structure on any given machine, as well as network or I/O performance, could be included in future versions of *dish*.

3.9 Interactive Commands

Some commands invoked by the shell expect their input and output channels to be connected to a terminal rather than a file, device, or socket. An example is a screen based editor like *vi(1)*. A line based editor, on the other hand, since it needs no special access to the terminal, works with the regular TCP sockets. If the remotely executing subtask does expect to talk to a terminal device, the server then sets up a pseudoterminal [*pty(4)*]—a feature introduced to the UNIX operating system along with TCP/IP to provide such remote login capabilities. It was required because many UNIX applications control the terminal or its device driver directly. For example, during login itself, the echo must be disabled during password entry to avoid showing it on the user's screen.

The client informs a server of this requirement with the `protocol` parameter in its job confirmation message. This parameter is a list of protocols for each open file descriptor. If a value is given as `pty`, then the server must set up the pseudoterminal and arrange to pass terminal control information to the client. Then it is required to use the *rlogin* protocol to control the remote interactive execution [34]. This method of remote execution provides an important improvement over the existing rlogin protocol because the entire *environment* space of the shell is available to the remote server. Note that the *rlogin* protocol referred to here is referring to its operational protocol, not the startup protocol since that is taken care of through the *dish* task initiation messages.

3.10 Files

Often shell pipelines extract their input from a file, or place their output in a file. This is accomplished using the *I/O redirection* facilities of the shell.

File names specified for input or output to *dish* are accessed only by the client in its local namespace. Arguments to commands, however, are passed through to the server to be handled exclusively by the subtask on the remote processor. Obviously,

a uniform name space across the network will make this more predictable and easier to deal with, but is not a strict requirement.

For a *dish* implementation to access files that are only available on a remote machine, it would require a mechanism of addressing those files. Something like the addressing method used in the Berkeley *r-commands* would be appropriate. For example, the command shown here would open the file through the *rcpd(8)* daemon on the host `st-brides`. Normally, `/tmp` is local to a workstation.

```
sort < st-brides:/tmp/inputfile
```

For files that a subtask expects to inherit from its parent, clients instruct the servers on which IP host and port each channel will be available (see Section 3.2.3). The client is then responsible for reading input from a specified file name (for shell input redirection using `<`), or placing output from server's subtask in output files (for output redirection using `>`).

3.11 Known Limitations

Although the *dish* protocol provides access to commands throughout the network, there are some drawbacks to a shell using this protocol as compared to the traditional single-processor version of the shell.

This version of the distributed shell protocol is somewhat operating system dependent, and currently is only defined for UNIX systems. Future versions of *dish* could be enhanced to allow for more operating system interaction. Open standards such as POSIX may make this a trivial extension to the protocol.

The elegance and efficiency of using the *pipe(2)* system call for setting up shell execution is lost. Instead, a distributed shell uses IP sockets for interprocess communication, whose setup is more complex and time consuming.

In the current implementation, an extra subtask is generated for every command pipeline to gather input and output for it. With a non-distributed shell, the client itself can assume this task.

Extra information must be gathered and maintained by the client shell. This includes tables of commands that should be executed locally for speed, and commands expected to control their terminal for remote execution.

The protocol design is dependent on fast networks and processors. If a network is slow or busy, the time spent waiting for *application* messages could become too long.

Finally, a distributed shell is impaired by its necessity to ensure an executable is valid before executing a likely looking file. Files must be opened and read to inspect their headers, slowing down the construction of hash tables and doing PATH checks.

Chapter 4

Implementation and Results

This chapter discusses the changes required to a shell needed to implement the *dish* protocol, and the proof of concept implementation. It also presents the results derived from it. A simple server and client were implemented to find acceptable parameters and evaluate its performance on two different networks.

4.1 Extension to Shell Function

Shells, desktops, and other command interpreters implementing the *dish* protocol will require a number of changes. These changes could be completely transparent to the user, but in order to gain full access to the potential benefits of a distributed shell, certain new commands, syntax, and variables should be introduced.

A distributed shell implementation should have the following features:

- a way to disable remote execution altogether.
- a way to flag individual commands to execute locally, remotely, or on a specific remote site.
- a way to control distribution of work: a specified group of hosts, a specified group of sub-nets, or a combination of the two.
- provide data encryption if available.

The following are the changes required in the shell implementation:

- Executable commands are now located over the network, so the algorithm for finding an executable must be modified to include the protocol.
- Input from and output to files, devices, or pipes must be arranged by the client, including an *I/O subtask* of the client.
- The shell must be able to run as a *server*, listening to the *dish* port instead of interacting with a terminal or shell script input.
- New data structures to keep track of hosts and remotely executing subtasks must be implemented.
- Checkpoint facilities should be provided.

Many optional features can be considered to make better use of resources over the network, and to give the user more control of remote execution should they require it. For example, a useful feature would be the ability to specify host names for executables, input, and output files, or characteristics required for specific commands (as provided by the Utopia shell [24]). For example, to run a command explicitly on host `samson-pk`, the command could look like:

```
samson-pk:/bin/cat stuff
```

To compile the file `test.c` on host `hobbema` with a compiler on any remote machine running the Linux operating system:

```
Linux:cc -o test hobbema:test.c
```

Characteristics can be compounded by simply concatenating the various desired strings separated by colons.

Such explicit specification would allow use of an "alias" mechanism to ensure that certain commands are always run locally, or on a processor with given characteristics or resources, instead of using internal tables.

To have a fully user-configurable execution environment, the shell should also have tables of commands with associated resources or requirements. These can either be set up by built-in shell commands invoked in startup files, or be contained in the user environment. For example, a user environment variable called `DISH_CMDLOCAL` can contain a list of all commands to run locally, and `DISH_CMDREMOTE` a list of all commands to run remotely. Associations between commands and other service characteristics can be stored in variables such as `DISH_CMDHOST`, which would have entries of the form `command,host1,host2,host3` where the first entry is the command name in question and the following entries are hosts that the user wishes to permit execute that command. An environment variable containing all hosts to advertise to should also be included, such as `DISH_SERVERS`.

The other primary modification of the shell involves changing its method of locating executables, starting subshells, and connecting their input and output channels, as described in Chapter 3.

4.2 Implementation Details

For the purposes of testing the protocol and deriving some timing information, a small shell was written using a subset of the *dish* protocol. The core command message types 0-4 were implemented (see Table 3.1). The remainder of the protocol is very similar to other systems and is already proven. Portions of the implementation specifically relating to the protocol are described here. The current implementation provides for task initiation over the network and provides pipe connections between subtasks using TCP sockets.

The shell first learns whether it is a server or a client. If a server, it opens a socket on which to listen to incoming service requests. If it is a client, it setups a command database, prints a prompt, and waits for command input from the user. The server runs until killed explicitly by a signal, because it is running as a daemon. The client runs until it reads an end-of-file, or it receives the `exit` built-in command.

4.2.1 Built-in Commands

Normally a shell has a great abundance of built-in commands. Any command that affects the internal state of the shell must be a built-in command. For example, the *cd(1)* command is internal to the shell since it must change the current working directory in the environment space. Sometimes commonly used but simple commands (e.g., *echo(1)*) are built-in to improve performance.

These commands are always run locally by the client, so the server does not need to know how to execute them. The prototype *dish* implementation has the following built-in commands. Optional arguments are enclosed in angle braces.

`exit` < *value* > Terminates the client. One optional parameter specifies the exit value for this shell.

`cd` *dir* Changes the working directory. This updates the value of the `PWD` environment variable also. It needs one parameter, the new directory name.

`setenv` *variable value* Sets an environment variable. It needs two parameters, the environment variable name and its new value.

`printenv` < *variable* > Prints out environment variable (name is first parameter). If no parameter is specified the entire environment is displayed.

`which` *command* Shows command from database, if any match the first parameter.

`rehash` Rebuilds command database. This also occurs if the `PATH` environment variable is changed using `setenv`.

The `setenv` command has other possible side effects, because some behaviour of the distributed shell is dependent on environment variable values. If the `PATH` environment variable is changed, the command database is rebuilt. For example, the variable `DISH_SERVERS` is the list of machines to send advertisements to. When this variable changes, the shell makes sure the new server names are valid by attempting to look up their IP addresses. This is done so that the name-to-address conversion

only takes place once instead of each time an advertisement is sent out. Environment variables are also used to determine timeouts. Currently, the only timeout defined is called `DISH_CLIENTWAIT` and is stored internally in microseconds.

4.2.2 Input Syntax

The prototype *dish* implementation has a subset of the usual shell functionality. Commands and other tokens are delimited by space characters and tabs.

Dish implements automatic command lookup—users do not need to specify the exact location of the programme they want to run, they need only type its name, and the shell will find the programme with the help of an environment variable called `PATH`.

File input and output redirection is supported. To obtain standard input from file `input`, and direct the results to `output`, the syntax used for *dish* is:

```
command <inputfile >outputfile
```

Command pipes are supported, indicated by a `|` character with white space on either side. Commands without pipes connecting their input and output can be separated by semi-colons or ampersands. A semi-colon (`;`) indicates sequential execution—the first command is completed execution before the second one can begin. An ampersand (`&`) indicates concurrent (background) execution. The command pipeline to the left of an ampersand is executed and put in the *background*. It will continue to run separately from the client shell.

Environment variables can be substituted in the input using the traditional `$` mechanism. For example, to see what is in the `PATH` variable, the following command can be executed:

```
echo $PATH
```

4.2.3 Command Database

A primary task of a command interpreter is to locate the commands the user wants to run. Some shells search the PATH environment variable for every search. Others, such as the C-shell [*csh(1)*], keep a hash table to speed command lookup.

If using such a database, a distributed shell needs to be more careful to place only valid executables in it. The C-shell, for example, hashes all the files contained in the directories specified in the `PATH`, assuming there are no extraneous files or subdirectories in those directories. If the user types the name of such an extraneous file, the shell attempts to execute it, receives an error, and fails. The distributed shell, on the other hand, uses the hash table to determine whether the client is capable of executing the command, which influences the urgency of finding a server to execute it instead.

The database is implemented as a hash table. The key is calculated using the command name as a seed. Entries contain the command name to verify matches, the directory that contains the executable, and a flag to indicate whether the directory it was found in comes after a "." entry in the `PATH`. If this flag is true, the shell must check the current directory for a similarly named executable that would then take precedence over the one in the database.

The database is not needed for the server because the client specifies a path with every job advertisement. If extensive `PATH`s were common, the server could maintain a database of directories that it commonly had to search. Such a database would have to be cleaned out regularly in case users with highly diverse requests were to fill it up with uncommon directories. The quick implementation used for this research simply searched the PATH for each job advertisement.

A problem encountered in the implementation was the amount of time required to build this database. If the shell was only required to scan through the directories and check the permission bits (via a call to *stat(2)*), the database could be built quickly. However, this implementation also opens each file and reads the first 128 bytes, and closes the file. The header is then examined to check if it contains a valid executable

format [*a.out(5)*], or if it contains an executable shell script. If it is a shell script and begins with the characters `#!`, then the name of a shell or interpreter programme is expected to follow on that line. The server must ensure that the programme specified is available. For example, a script starting with `#!/usr/bin/perl` can only run on a system with a perl executable in that location. If required, a distributed shell could do without the database, but it would increase the response time for each pipeline.

4.2.4 Command Pipelines

Once the shell has parsed a command pipeline, it must take some extra steps not normally required in a non-distributed shell.

From left to right, the shell takes each command and checks environment variables for information regarding that command. This information is used for creating subtasks later, including whether to execute the command locally or what kind of processor must be used.

Once all the commands are checked in the database, an I/O subtask is spawned (see Section 4.2.7) to gather input for the first command in the pipeline and to collect output from the last command (in fact, they may be the same process). This subtask is given the first two assigned port numbers, which will later be used to connect the input channel of the first command in the pipeline and the output channel of the last command. This implementation is limited to the first two channels, the standard input and output. The standard error channel is lost for server tasks and prints to the terminal of the client for all tasks executed by the client itself.

Next the advertisement messages are built and sent to the hosts listed in the environment variable `DISH_SERVERS`. Applications are received, and a server is chosen. The client takes any tasks that are unassigned. For the purposes of this experiment, the algorithm for choosing a server was simple: choose the first server that responds capable of doing the job. In practise this method works well, since the fastest and nearest computers will be the first to reply, unless they are already too busy, in which case they will not be first. Other limitations, such as processor classes, are simulated

by placing a directory in the `PATH`, which is local on all machines (`/tmp` was used), and populating it with various commands on the different participating servers.

As each command is assigned, it is given a new port number for its input and output channels, and the host name for its input channel. (In this implementation, the client chooses all port numbers in a certain range known to be safe on the given network. In a portable implementation, the scheme described in Section 3.2.3 must be used.) This host name is the host of the client for the first command in the pipeline, and the name of the host awarded the previous command for the rest of the commands. This implementation uses the simplification of awarding commands in left-to-right order, but there is no inherent reason for the client to award jobs in this manner. To award them in any other order would be more complicated because for each connected command, one server must know the host name of the other server to which it must connect.

Once the subtasks are spawned, and if the pipeline is running in the foreground, the client awaits the completion of the I/O subtask started previously to the bidding process. Finally, the client waits for each command in the pipeline to send its *completion* message, unless they are running in the background. Background tasks are left to execute independently of the client *dish*, but the client still keeps track of the incoming completion messages and its locally running subtasks so it can properly clean up if they complete.

4.2.5 Server Operation

This version of the *dish* server is implemented as a user-level daemon. No authentication is needed because each user must run their own daemon on the specified server nodes to get any response. For a full implementation, the daemon would have to do the authentication in response to each advertisement. Then it would set its user identification [*setuid(2)*] to that of the advertisers before invoking the subtask. If the daemon is invoked by a super-daemon [*inetd(8)*], then it must stay active until the subtasks it invoked are complete.

The server, instead of reading commands from the standard input, ignores its input channel, (a normal daemon *disassociates* itself from its controlling terminal and usually is started at boot time), and assigns a socket to listen to the *dish* port.

If a message comes in, the server parses it and if it is an advertisement, it immediately checks through the path for the requested command. If it finds an executable, it constructs an *application* message and sends it back to the client. It then returns to its original state.

If the received message is a job *offer*, it then extracts all pertinent information from the offer, sends a *confirmation*, and spawns the subtask required.

4.2.6 Execution of Subtasks

The client and server use the same code for execution of subtasks. The algorithm is similar to what might be expected for a shell implementation. First a `fork` system call is made, resulting in two processes. The parent process waits for the completion of the child, unless it is the client and the task is marked as a background task. The server, by waiting for the child task to complete, prevents any other *dish* clients from reaching this machine. A different implementation would be required for a multiprocessor node to enable more than one concurrent remote *dish* task, or for a seriously unbalanced set of nodes where some servers could take higher loads of subtasks.

The child process from the `fork` call first sets up its input and output pipes. The process actively connects to any channels that have hosts specified in their `hosts` field of the *offer* message. Then, it passively waits for another process to connect to the channels not specified in the `hosts` parameter. In the current implementation, it is assumed that the input is always specified and the output is never specified, although this is simply for convenience and has no impact on the functionality of the service. Once the pipes have successfully connected, the child process does an `execv` system call [*exec(2)*] to invoke the command that was requested.

4.2.7 Input/Output Subtask

The *dish* client invokes a special subtask for each pipeline called an *I/O subtask*. This process is responsible for providing input to and gathering output from the pipeline processes. Unlike the command subtasks, which open their active sockets first and then their passive ones, this module must do it in the other order, since the first programme in the pipeline is expecting to connect immediately actively, and the last programme will be passively awaiting its connection to the I/O subtask last (See Figure 3).

In a more general implementation, both sides would have to poll for both active connections and passive, since it would not always be guaranteed that the process to receive or provide a connection would be there at any given time.

Once the connections are made, this process simply reads from the standard input it inherited from the client (usually the keyboard), and sends it to the first task in the pipeline, while simultaneously reading from the last task in the pipeline and sending it to the standard output (usually the user's screen). To do both tasks in one process, the *select(2)* call is used to block for input on both channels.

Implementation of the `pty` option for the `protocol` parameter described in Section 3.2.3 requires the I/O process to set up a pseudoterminal and communicate any signals generated by its input to the other tasks in the pipeline.

The I/O subtask exits after both its input and output are finished. Reads and writes on sockets return the number of bytes successfully read or written, or a negative error value, or zero to show an end-of-file condition. An end-of-file on a socket means the other end of the connection has closed, so the subtask can assume that communication channel is done and close down the four sockets it was using, and exit.

An implementer may wish to provide a flag to tell the *dish* client to make a circular pipeline by avoiding the I/O subtask altogether and telling the last process to connect to the first process. This could be useful for some types of distributed applications.

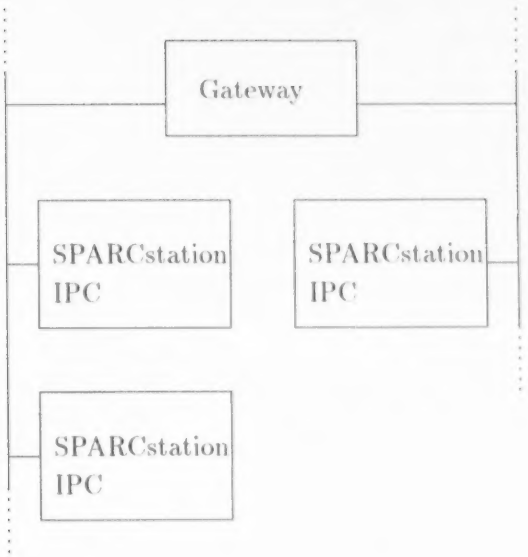

Figure 4: Network 1

4.3 Performance of Distributed Shell

The distributed shell was tested on two different networks consisting of Sun SPARCstations, Intergraph Clipper workstations, IBM RISC System/6000 workstations, and Digital Equipment Corporation (DEC) AXP workstations. They were connected with standard (10 Mbit/s) ethernet.

The first network tested was a large, busy, TCP/IP network using gateways. A set of three SPARCstation IPC's running SunOS 4.1 were used, two on the same ethernet wire, one separated by a gateway. The partial network is shown in Figure 4. The client was set up with an *application timeout* of 100 milliseconds.

To measure performance of the startup protocol, real time was measured before and after all code in the client involving the *dish* protocol, and accumulated for each pipeline execution. The amount of time measured this way for the entire pipeline is called the *distribution overhead.* The average distribution overhead for starting up a single remote command was approximately 700 milliseconds. For two remote

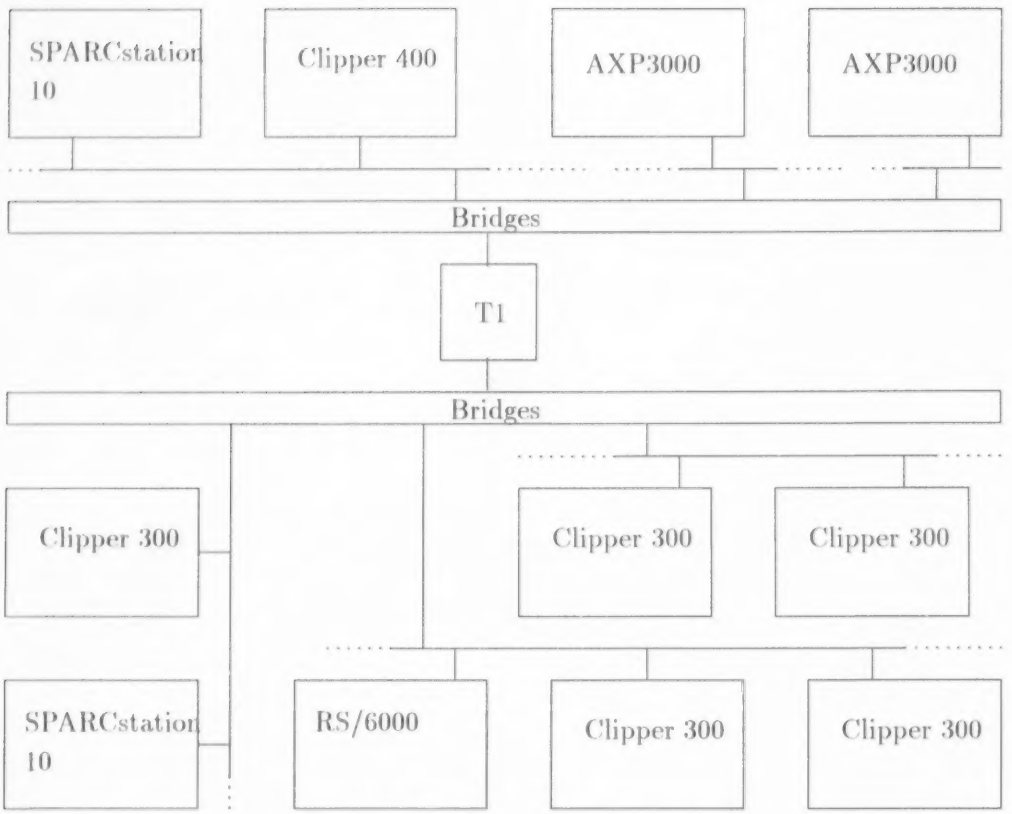

Figure 5: Network 2

commands, the time to start up a job rose to 1.4 seconds. For three commands, it took approximately 2.1 seconds.

This increase in time is due to the implementation rather than the protocol. The current implementation only advertises a new job after it has assigned the previous one. A full implementation would broadcast all advertisements at once. Therefore, the overhead for the protocol is within one order of magnitude of the delay required for remote servers to reply to *dish* job advertisements.

When the application timeout on this network was reduced to 50 milliseconds, the overhead was reduced also by approximately half. However, even when they are not separated by a gateway, the servers sometimes cannot reply in only 50 milliseconds.

The second network tested was a larger mixed protocol network using bridges and repeaters instead of gateways. The machines on the network used for testing are shown in Figure 5. (Repeaters transmit all packets they get on one network to another network. Bridges do not retransmit packets aimed at a target that it knows is on the same network the packet originated on.) This trial was performed with two sets of processors located in different sites approximately 7km apart. They were connected by local ethernet wires, several repeaters, bridges, and a T1 (1.45 Mbit/s) link. This trial was run when the machines and network were otherwise idle. One site consisted of two DEC AXP workstations with Alpha processors running OSF/1 (these are substantially faster computers than the others), a Sun Sparcstation 10 running SunOS 4.1, and an Intergraph Clipper 400 workstation running CLIX 6.0. Another Sparcstation 10, an IBM RS/6000 running AIX 3.1, and five Clipper workstations running CLIX 6.0 were on the other side of the T1 link. At each local site, some stations were connected to the same ethernet wire, others were connected through repeaters and bridges.

The following results were observed when using the 100 millisecond application timeout. One Sparcstation was the client. The other Sparcstation, a server, was always the fastest to respond to a job advertisement, and the average overhead incurred was about 400 milliseconds. The other stations, regardless of their proximity on the network, increased the overhead for a command execution to 600-700 milliseconds.

On the second network, 100 milliseconds was the minimal value for the wait period for job applications, and often with a smaller value no servers could reply in time. Furthermore, eliminating servers did not affect the timing results. This is because of the nature of the job advertisement cut-off, and because the traffic of the job advertisements and applications is negligible on these networks. Furthermore, the amount of network hardware separating the computers seemed to have little impact on the distribution overhead. The distribution overhead does not seem to correspond to proximity or processor performance. In fact, access to the hard disc and network hardware performance probably make the most impact on the overhead.

Note that this implementation did not attempt any kind of optimisation on the code or algorithms used. A full implementation, outside the scope of this thesis, could improve performance by tuning the algorithms used. Another important avenue for improving performance would be to make the *application timeout* a more dynamic variable, self-adjusting to the network conditions. For example, if applications are regularly arriving within a much shorter time frame than the current value, it should be decreased to reduce apparent delays. Furthermore, optimal response could be acheived by always timing out after the first available (and therefore probably most desirable) server has replied. Finally, optimising the command lookup in the *dish* server would also help to reduce the overhead.

Chapter 5

Conclusion

This thesis demonstrates the feasibility of implementing a distributed shell using a high-level protocol, providing better access to processing resources on a network and moving one step toward the ideal of distributed computing—network transparency.

The *distributed shell protocol* was designed to take advantage of fast, high bandwidth networks and high performance workstations. It achieves load-balancing as part of the server query process, instead of keeping resource availability databases that are quickly out of date or else require substantial resources to maintain. It provides interactive and transparent access to all commands on all servers to the user, without requiring knowledge of their architectures or resource availability.

The trial implementation of the protocol shows that with current network standards, a slight delay would be noticable when using a distributed shell. However, a production implementation would involve substantial work on optimisation of response time from *dish* servers, bringing it closer to the time allowed for servers to respond (currently 100 milliseconds).

Interesting further work would be to implement a fully distributed and optimised shell with all the features described in Chapter 3, including checkpointing facilities, fully interactive remote execution using pseudo-terminals, and signal transmission. A broadcast service could be installed to find out how it affects performance in general and what impact it has on network load. Further analysis of the protocol design would

include a breakdown of the distribution overhead into command message transmission, message construction, server selection, and location of executables. Also, it would be useful to have a benchmark to find how much slower TCP connections over the network are than the traditionally implemented *pipes* of a non-distributed shell.

This protocol fills a gap between the kinds of distribution tools provided with UNIX systems at the shell level, specialised distributed programming paradigms such as Enterprise, and scripting tools such as *Glish*, which usually offer only limited distribution facilities. It also provides a more generalised approach to distributing shell subtasks than the load-sharing shell provided with the *Utopia* system. It allows distribution of tasks in a paradigm that is familiar to most users of the UNIX system, with minimal changes to scripts, yet allows much more flexible use of the shell methodology by providing checkpointing and two-way pipes.

Bibliography

[1] Sun Microsystems Inc. Request for comments 1057: Sun RPC. Network Information Center, June 1988. [nic.ddn.mil:/rfc/rfc1057.txt].

[2] K. Birman. The process group approach to reliable distributed computing. Technical Report TR-91-1216, Cornell University, July 1991. [ftp.cs.cornell.edu:/TR-91-1216.ps.Z].

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual*, May 1993.

[4] Douglas E. Meyer and Ralph W. Wilkerson. HIGHLAND: A graph-based parallel processing environment for heterogeneous local area networks. In *The Fifth Distributed Memory Computing Conference*, pages 742–747, April 1992.

[5] Jonathan Schaeffer, Duane Szafron, Greg Lobe, and Ian Parsons. The Enterprise model for developing distributed applications. *IEEE Parallel and Distributed Technology*, October 1993. to appear.

[6] A. Singh. *A Template–Based Approach to Structuring Distributed Algorithms Using a Network of Workstations.* PhD thesis, University of Alberta, 1992.

[7] J. Reynolds and J. Postel. Request for comments 1340: Assigned numbers. Network Information Center, July 1992. [nic.ddn.mil:/rfc/rfc1340.txt].

[8] Ellie Cutler, Daniel Gilly, and Tim O'Reilly. *The X Window System in a Nutshell.* O'Reilly & Associates, Sebastopol, California, 1992.

[9] Douglas E. Comer. *Internetworking with TCP/IP*, volume 1. Prentice-Hall, Inc., 1991.

[10] Larry Wall and Randall Schwartz. *Programming Perl.* O'Reilly & Associates, Sebastopol, California, 1991.

[11] *UNIX Reference Manual.*

[12] D.M. Ritchie and K. Thompson. The UNIX time sharing system. *Communications of the ACM*, 17(7):365–375, 1974.

[13] John K. Ousterhout. *An Introduction to TCL and Tk (DRAFT).* Addison Wesley, 1993.

[14] Open Software Foundation. *White Papers on the Distributed Computing Environment.* O'Reilly and Associates, Inc., 1991.

[15] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP*, volume 3. Prentice-Hall, Inc., 1993.

[16] Sun Microsystems Inc. *STREAMS Programming Manual*, March 1990.

[17] Kai Hwang, William J. Croft, George H. Goble, Benjamin W. Wah, Faye A. Briggs, William R. Simmons, and Clarence L. Coates. A UNIX-based local computer network with load balancing. *IEEE Computer*, pages 55–65, April 1982.

[18] T.A. Marsland, T. Breitkreutz, and S. Sutphen. A network multi-processor for experiments in parallelism. *Concurrency: Practice and Experience*, 3(3):203–219, June 1991.

[19] Sun Microsystems Inc. Request for comments 1014: XDR: External data representation standard. Network Information Center, June 1987. [nic.ddn.mil:/rfc/rfc1014.txt].

[20] Vern Paxson and Chris Saltmarsh. *Glish:* a user-level software bus for loosely-coupled distributed systems. In *USENIX Winter Conference*, 1993.

[21] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4), June 1989.

[22] Don Libes. Kibitz—connecting multiple interactive programs together. *Software—Practice and Experience*, 23(5):465–475, May 1993.

[23] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. Technical Report CSRI-257, Computer Software Research Institute, University of Toronto, April 1992. [sys.toronto.edu:/white-technical-reports/257].

[24] P. Delisle and Jingwen Wang. Load sharing tcsh (lstcsh) user's manual. Obtained over Internet, September 1991.

[25] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the UNIX kernel. In *Usenix Winter Conference*, March 1992.

[26] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *ACM Operating Systems Review*, 27(2):72–76, 1993.

[27] H.E. Bal. *The Shared Data-object Model as a Paradigm for Programming Distributed Systems.* Mathematisch Centrum, Amsterdam, 1989.

[28] S.J. Mullender and A.S. Tanenbaum. The design of a capability-based distributed operating system. *Computer Journal*, 29(4):289–299, August 1986.

[29] George A. Champine, Daniel E. Geer, and William N. Ruh. Project athena as a distributed computer system. *IEEE Computer*, pages 40–50, September 1990.

[30] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Proceedings*, March 1988.

[31] Li Gong. A security risk of depending on synchronised clocks. *Operating Systems Review*, 26(1):49–53, January 1992.

[32] A. Kehne, J. Schönwälder, and H. Langendörfer. A nonce-based protocol for multiple authentications. *ACM Operating Systems Review*, 26(4):84–89, October 1992.

[33] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[34] B. Kantor. Request for comments 1282: BSD rlogin. Network Information Center, December 1991. [nic.ddn.mil:/rfc/rfc1282.txt].

Appendix A

Dish Message Format

Dish messages are datagram packets with the following format. All messages consist of printable ASCII characters. The first five characters, for version zero of *dish*, are: "`dv=0\n`" where "`\n`" is a newline character. After the fifth character are fields separated also by newline characters. Fields consist of an identifier, an equal sign (`=`), and a value. They may be placed in any order, except for the `env` field, which must be last because it may have newlines in it. A *dish* control message is self-contained, unless it contains an `env_count` parameter, which indicates the number of messages used to transmit the entire environment. All messages must contain a `type` field.

Following is a list of all message parameters defined for version zero of the distributed shell protocol:

`arch` The processor type of a server.

`argc` The argument count for a sub-task.

`argv` The arguments for a sub-task. Each argument is enclosed in square braces.

`client_port` The IP port number for the server to reply to.

`command` The command for a subtask.

`cpuperf` The processor performance of a server.

`cwd` The current working directory for a subtask.

`dv` The *dish* Version number (currently always 0)

`encryption` Encryption scheme for a intertask communication.

`env_count` The number of datagrams used to provide a environment description.

`env` The environment. This must be the last parameter because it will probably contain newlines.

`hosts` List of hosts which server must actively connect to to get descriptors for subtasks. Separated by colons.

`host` The host of the originator of the message.

`interval` The interval desired between checkpoint messages, format is seconds and microseconds in decimal separated by a colon.

`ioperf` The io performance measure of a server.

`job_id` The job identification string. Consists of client host name, client process id, and a sequential number separated by periods.

`memsize` Real memory size needed by task or available on a server.

`netperf` Network performance of a server.

`os` Operating system running on a server or desired by client.

`path` List of directories in which to look for the command. Separated by colons.

`ports` List of port numbers chosen by server to connect incoming interprocess communication channels.

`processors` Number of processors of a server.

`protocols` Protocol list to use for intertask communication. Separated by colons.

`signal_id` Identification of signal transaction.

`signal` Which POSIX signal number to send to subtask or client.

`status` The return status of a subshell.

`ticket` The authorisation key for a client.

`timestamp` The timestamp (seconds and microseconds from epoch, separated by a colon) of a checkpoint.

`type` Type of message. See list above.

`umask` Umask of client. [*umask(2)*].

`user` Userid which client is running for.

`vmemsize` Size of virtual memory on a server.

Appendix B

Example Startup Messages

The following messages show an example transaction between two hosts, `st-brides` and `sundog`, where a server on `sundog` performs the command `ls -lt` for the client running on `st-brides`.

Advertisement

```
dv=0
type=0
user=tim
job_id=st-brides.1938.23
client_port=22783
command=ls
cwd=/home/sundog/tim/xyz
umask=022
path=.:/home/sundog/tim/bin:/bin:/usr/bin
host=st-brides
arch=sun4m
os=SunOS
```

Application

```
dv=0
type=1
job_id=st-brides.1938.23
host=sundog
```

Offer

```
dv=0
type=2
job_id=st-brides.1938.23
argc=2
argv=[ls] [-l]
ports=22784::22785
hosts=st-brides::st-brides
env=SHELL=/bin/dish
HOME=/home/sundog/tim
PWD=/home/sundog/tim/xyz
TERM=vt100
LOGNAME=tim
PATH=.:/home/sundog/tim/bin:/bin:/usr/bin
```

Confirmation

```
dv=0
type=3
job_id=st-brides.1938.23
host=sundog
ports=:27827:
```

Completion

```
dv=0
type=4
job_id=st-brides.1938.23
status=0
```

⋈